

APPRENEZ LE JAVASCRIPT

DYNAMISEZ VOS PAGES WEB !

1. JAVASCRIPT : PRÉSENTATION ET MISE EN PLACE	4
1.1 HISTORIQUE	4
1.2 ENVIRONNEMENT DE DÉVELOPPEMENT	4
1.3 TESTER L'ENVIRONNEMENT DE DÉVELOPPEMENT	6
2 LES BASES DU JAVASCRIPT	9
2.1 LES VARIABLES	9
2.2 LES TYPES DE VARIABLE	13
2.3 LES OPÉRATEURS	22
3 CONTRÔLE DU FLUX	29
3.1 LA CONDITION : IF	29
3.2 LE CHOIX : SWITCH	29
3.3 LA BOUCLE FOR	30
3.4 LA BOUCLE FOR...IN	30
3.5 LA BOUCLE FOR...OF	30
3.6 LA BOUCLE WHILE	31
3.7 LA BOUCLE DO .. WHILE	31
3.8 L'INSTRUCTION CONTINUE	31
3.9 L'INSTRUCTION RETURN	31
4 TABLEAUX	32
4.1 PRINCIPE	32
4.2 ÉCRITURE LITTÉRALE	32
4.3 ACCÈS AUX ÉLÉMENTS	32
4.4 PARCOURS	33
4.5 QUELQUES MÉTHODES UTILES	33
5 LES FONCTIONS	36
5.1 PRINCIPE	36
5.2 DÉFINITION DES FONCTIONS	36
5.3 VALEUR DE RETOUR	38
5.4 PARAMÈTRES ET ARGUMENTS	38
5.5 CONTEXTE D'EXÉCUTION	41

6	LES OBJETS	44
6.1	NOTATION	44
6.2	MÉTHODES	44
6.3	INSTANCIATION PAR CONSTRUCTEUR	45
6.4	PROTOTYPES	46
6.5	INSTANCIATION PAR PROTOTYPE	52
6.6	PLUS LOIN AVEC LES PROTOTYPES : HÉRITAGE	53
6.7	LES CLASSES	57
7	PROGRAMMATION AVANCÉE	61
7.1	LE DÉBOGAGE	61
7.2	API, PLUGINS ET FRAMEWORKS	64
8	DOCUMENT OBJECT MODEL	65
8.1	INTRODUCTION	65
8.2	ACCÉDER À LA STRUCTURE DU DOM	67
8.3	MODIFIER LA STRUCTURE DU DOM	73
9	MODIFIER LE CSS	81
9.1	OBTENIR LE STYLE D'UN ÉLÉMENT	81
9.2	MODIFIER LE STYLE D'UN ÉLÉMENT	81
10	LA GESTION ÉVÈNEMENTIELLE	82
10.1	QU'EST-CE QU'UN ÉVÈNEMENT ?	82
10.2	RAPPEL SUR LE FOCUS	82
10.3	ÉCOUTEURS	82
10.4	LES DIFFÉRENTS ÉVÈNEMENTS	84
10.5	CAPTURE ET BOUILLONNEMENT	89
10.6	MODIFIER LE COMPORTEMENT DES ÉVÈNEMENTS	90
10.7	RAPPEL SUR LE CLONAGE	91
11	LA GESTION DES FORMULAIRES	92
11.1	LES PROPRIÉTÉS DES BALISES DE FORMULAIRE	92
11.2	LES MÉTHODES SPÉCIFIQUES AUX FORMULAIRES	95
11.3	VALIDATION D'UN FORMULAIRE	96
12	ANIMER LES PAGES WEB	99
12.1	GESTION DU TEMPS	99

12.2	ANIMER DES ÉLÉMENTS	99
13	AJAX	101
13.1	QU'EST-CE QUE C'EST ?	101
13.2	XHR	102
14	DES PROMESSES, TOUJOURS DES PROMESSES	111
14.1	LES PROMESSES EN JAVASCRIPT	111
14.2	L'API FETCH	115
14.3	LES PROMESSES AVEC ES7	115

1. JavaScript : présentation et mise en place

1.1 Historique

- 1995 : la société Netscape Communication Corporation décide qu'elle a besoin d'un langage permettant d'exécuter du code sans utiliser le serveur, directement dans le navigateur du client. Elle missionne Brendan Eich pour développer ce langage, et lui donne deux semaines pour aboutir ! Cet ingénieur s'inspire de langages existants (comme Java) et réussit incroyablement à boucler le projet en dix jours seulement. Le langage, initialement appelé LiveScript est finalement renommé JavaScript d'une part pour surfer sur la vague du Java, langage en pleine expansion de Sun Microsystems partenaire de NCC à l'époque, et d'autre part pour faire de la publicité à Java.
- 1996 : JavaScript est intégré au navigateur Netscape Navigator 2.0 en mars. Le succès est immédiat. À tel point que Microsoft réagit immédiatement en proposant son propre langage JScript dans son navigateur web Internet Explorer 3 en août. En novembre, NCC répond en soumettant son langage à ECMA International (European Computer Manufacturers Association) pour standardisation.
- 1997 : les travaux de l'ECMA aboutissent en juin pour proposer le standard ECMA-262 aussi appelé ECMAScript (ES), qui est donc le standard de JavaScript.

Suivent diverses évolutions du langage ES jusqu'à la dernière vraie révolution en juin 2015 avec ECMAScript 6 (ES6), aussi appelé JavaScript 2015. Cette version apporte bon nombre de nouvelles possibilités, dont un rapprochement au niveau du vocabulaire vers les langages objets à classe (comme Java). JS continue à utiliser le prototypage mais introduit la notion de classe pour simplifier l'accès du langage aux personnes qui viennent des langages objets à classe.

Depuis 2015, ECMAScript a une nouvelle version tous les ans. La dernière version (ES 11 en 2020) apporte quelques nouveautés par rapport à ES6, mais somme toute rien qui n'impacte les habitudes de développement pour un débutant. Nous resterons donc essentiellement sur ES6.

1.2 Environnement de développement

1.2.1 Éditeur de texte

Pour écrire le code source, il nous faut un éditeur de texte, et rien d'autre, comme pour tous les langages de programmation. Attention, Microsoft *Word* n'est pas un éditeur de texte, pas plus que LibreOffice *Writer*. Je parle bien ici d'un outil comme le *NotePad* sur Windows.

Cependant, certains outils sont bien mieux adaptés au codage que d'autres. En particulier, de nombreux utilitaires permettent de :

- Colorer les mots du langage, ce qui permet de mieux les reconnaître,
- Auto-complémenter le code (fermer les parenthèses ou les accolades, ou même les mots du vocabulaire du langage) pour éviter les fautes de syntaxe,
- Exécuter le code sur un serveur local.

Bref, de vrais outils bien utiles. En particulier pour JS je vous propose d'utiliser le logiciel *Visual Studio Code* (VSC) fonctionnant sur Linux, Mac et Windows, à télécharger sur <https://code.visualstudio.com/download>. Vous devez déjà avoir ce logiciel sur votre machine virtuelle.

VSC est très simple et très rapide... et est codé en JavaScript ! Il est également facilement améliorable grâce à des extensions. Je vous propose d'installer l'extension *Beautify* (de *HookyQR*) qui permet

d'indenter correctement le code à la sauvegarde. Pour ce faire, allez dans le menu *View -> Extensions* ou cliquez sur l'icône correspondante dans le menu latéral (entouré de rouge sur cette capture) :



, puis dans la partie qui s'ouvre recherchez *Beautify* :



Enfin cliquez sur le bouton *Install*.

De la même façon, nous allons installer l'extension *Live server* (de *Ritwick Dey*) qui permet de visualiser une page HTML directement en créant un serveur local.

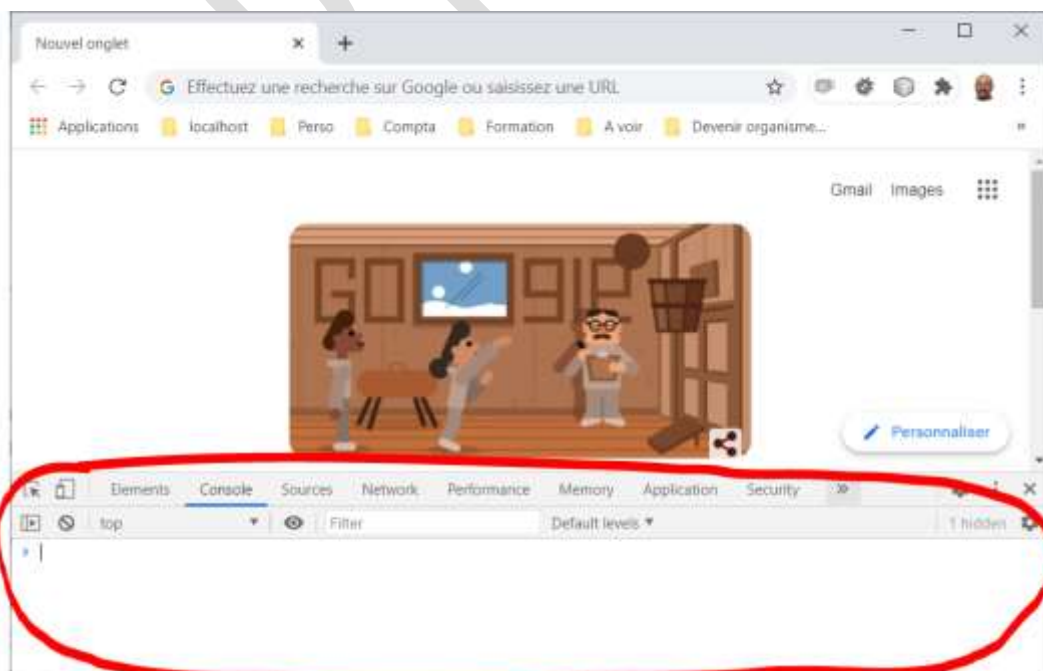
Nous voilà bien outillés pour commencer à coder sereinement.

Notez cependant que vous pouvez tout à fait utiliser un monstre comme *Eclipse* ou *NetBeans* pour coder en JS, ça marche aussi... C'est juste plus lourd 😊.

1.2.2 Navigateur web

Ensuite il nous faut un interpréteur JavaScript. Ça tombe bien, il y en a un dans tous les navigateur web ! Que vous utilisiez *Firefox*, *Chrome*, *Opera*, *Safari*, *Edge* voire même *Internet Explorer* (mais qui utilise encore ça de nos jours ?), vous avez un interpréteur JavaScript. Ce qui, au passage, fait de JavaScript le langage le plus interprétable dans le monde.

Nous verrons que JavaScript permet d'interagir avec les pages web, mais au début nous n'utiliserons que la console de l'interpréteur. Pour l'afficher, il suffit dans la majorité des cas d'appuyer sur la touche *F12* et on obtient en bas de fenêtre une interface du genre (sur Chrome) :



Il n'y a plus qu'à cliquer sur l'onglet *Console* de cet outil, et nous voilà dans la console.

1.2.3 Sites de test

Un dernier outil bien pratique parfois, ce sont les sites bac-à-sable. Ce sont des sites dans lesquels vous pouvez taper votre code HTML, et votre code JavaScript, et qui vous montrent en direct le résultat. Surtout utile lors de développements web, parce que la console n'est pas toujours très fonctionnelle dans ces sites. Je vous propose d'aller voir les plus connus : [JSFiddle](#), [JS Bin](#) et [CodePen](#).

1.3 Tester l'environnement de développement

Il est d'usage, lorsqu'on découvre un langage de programmation, de lui faire écrire *Hello World!* Nous n'allons pas déroger à la règle, ce serait inconvenant, et ça nous permettra de voir si notre environnement de développement est bien installé.

1.3.1 Dossier de travail

Créez un sous-dossier `PremiersPas` dans lequel vous allez créer un répertoire `js`. Vous pouvez faire ça directement dans VSC dans la partie navigateur de fichiers à gauche (icône du haut).

1.3.2 Script

Il nous faut du code à interpréter. Créez un fichier `HelloWorld.js` dans le sous-répertoire `js`, et mettez-y le code suivant :

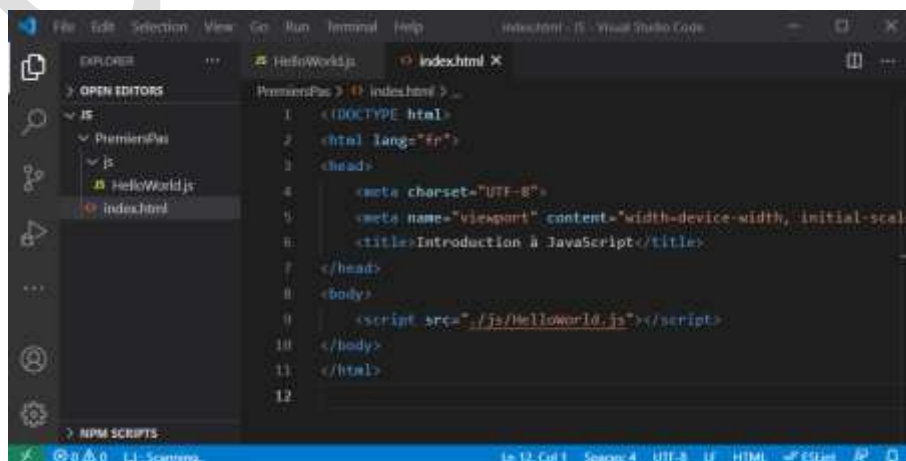
```
console.log("Hello World!");
```

1.3.3 Page HTML

Notre code sera interprété par un navigateur web. Il nous faut donc une page web. Créez donc un nouveau fichier dans le répertoire `PremiersPas` et appelez-le `index.html`, puis mettez ce code dedans :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Introduction à JavaScript</title>
</head>
<body>
  <script src="./js/HelloWorld.js"></script>
</body>
</html>
```

Vous devez donc avoir quelque chose comme :

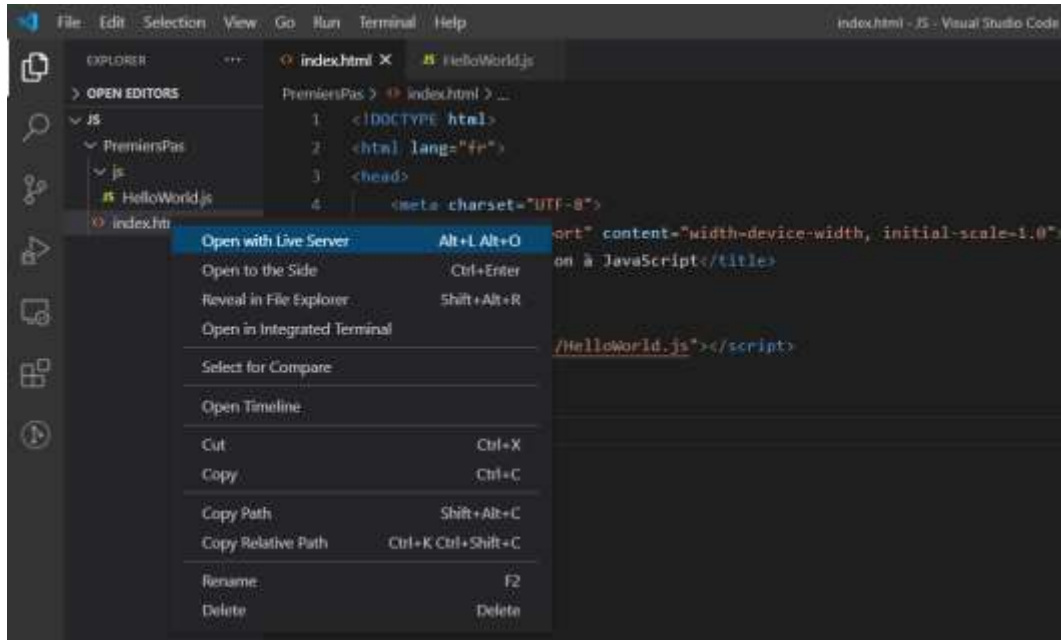


1.3.4 Afficher la page

Il ne reste plus qu'à afficher la page web. Ouvrez votre navigateur et ouvrez le fichier (**CTRL+O**) ou tapez l'URL (adaptez selon votre propre chemin d'accès au fichier) :

`file:///home/stag/JavaScript/PremiersPas/index.html`

Mieux : clic droit sur le fichier `index.html` et choix de *Open with Live Server* :



Et on constate ? Rien ! Nada ! Mais que se passe-t-il ?

Rappelez-vous, nous avons dit que pour l'instant nous allons afficher des choses dans la console de l'interpréteur ! Il nous faut donc afficher cette console (**F12**) et peut-être recharger la page (**F5**). Et là, miracle, notre message apparaît.

1.3.5 Analyse

Qu'avons-nous fait ? Nous avons créé une page web `index.html` qui contient l'appel à un fichier `HelloWorld.js` qui contient le code source qui a été interprété par le navigateur.

Ce code ne contient qu'une seule ligne, mais nous allons toutefois l'analyser :

```
console.log("Hello World!");
```

- `console` est un objet qui existe toujours dans l'interpréteur du navigateur, nous pouvons donc l'utiliser.
- De cet objet, nous allons utiliser un traitement qui s'appelle `log` qui a pour fonctionnalité d'afficher un message dans la console.
- Ce message est donné entre les parenthèses et les guillemets (`"Hello World!"`). Le traitement `log` doit donc afficher ce qui se trouve à l'intérieur des guillemets. Et c'est bien ce qu'on trouve dans la console : *Hello World!*

Nous aurions pu utiliser la fonctionnalité d'affichage du texte dans une fenêtre surgissante (pop-up) en remplaçant `console.log` par `alert` :

```
alert('Hello World!');
```

Cependant, l'usage intensif des pop-ups est fatigant et nécessite de multiples clics, nous nous contenterons donc, pour l'instant, d'utiliser l'affichage en console.

Maintenant que nous avons un environnement fonctionnel, nous allons pouvoir décrire le langage JS.

COPYRIGHT

2 Les bases du JavaScript

JavaScript est un langage impératif, c'est-à-dire qu'on donne des instructions à la machine qui doit les exécuter. Une instruction est un ordre simple donné à la machine. Ça peut être *écrit quelque chose à l'écran*, ou encore *calcule 3+5* par exemple.

Une instruction tient normalement sur une seule ligne. On peut en mettre plusieurs sur une même ligne à condition de les séparer par des points-virgules. Il est tellement couramment admis qu'il faut mettre un point-virgule en fin de ligne, même si cette ligne ne contient qu'une seule instruction, que c'est presque devenu une obligation. Ce n'est pas obligatoire en JavaScript, toutefois. Mais faites-le quand même, sous peine d'effets de bords indésirables dans certaines situations.

Les instructions peuvent être regroupées en bloc si on les entoure d'accolades :

```
{  
  instruction1;  
  instructions2;  
  ...  
  instructionsN;  
}
```

Un bloc d'instructions est considéré comme une instruction unique mais ne nécessite pas de point-virgule après l'accolade fermante.

2.1 Les variables

2.1.1 Déclaration et affectation

Une variable est un espace réservé dans la mémoire de l'ordinateur qui permet de stocker une information. Cette information peut être de différents types, que nous allons détailler. Par analogie, imaginez la mémoire de l'ordinateur comme un immense entrepôt. La variable est une boîte qui contient une information déposée quelque part dans cet entrepôt.

Une variable est définie par trois éléments indispensables :

- Un identificateur (ou un *nom*) : par analogie, l'identificateur est l'étiquette qu'il y a sur la boîte, pour la reconnaître parmi toutes les autres boîtes dans la mémoire. L'identificateur d'une variable obéit à certaines règles de la grammaire JavaScript, qui sont décrites plus loin.
- Un type : le type d'information que peut contenir la variable.
- Une adresse : elle définit l'endroit où se trouve la boîte dans l'entrepôt.

Dans certains langages, le type est nécessaire, dans d'autres non. Les langages qui nécessitent de déclarer le type systématiquement sont dits à typage statique ou typage fort. Ceux qui acceptent qu'on ne dise rien et qui déterminent par eux-mêmes le type d'information sont dits à typage dynamique ou typage faible. JavaScript est un langage à typage dynamique, on peut donc utiliser une variable sans indiquer son type. Si ça peut-être un avantage, ça s'avère également être un inconvénient d'importance, que nous verrons lorsque nous aborderons le *transtypage*.

Pour ce qui est de l'adresse, c'est complètement transparent pour l'utilisateur, c'est l'ordinateur qui gère cet aspect.

Une variable doit en premier lieu être déclarée, c'est-à-dire que nous allons indiquer à l'ordinateur que nous voulons réserver un espace mémoire (une boîte) pour pouvoir y stocker de l'information temporairement. Cette déclaration se fait avec le mot clé `var` :

```
var solution;
```

Cette ligne indique que je veux une boîte que j'appelle `solution`. Je pourrai dorénavant utiliser cette boîte simplement en donnant son identificateur.

Une fois cette déclaration faite, on peut utiliser la boîte pour y stocker l'information. Ça s'appelle affecter une valeur à la variable. L'affectation de valeur se fait avec le signe `=` :

```
solution = 25;
```

On peut déclarer et affecter une variable en même temps. Les deux exemples précédents peuvent se simplifier en :

```
var solution = 25;
```

Que se passe-t-il si j'affecte plusieurs fois des valeurs à la même variable ? C'est bien simple, chaque fois que vous affectez une valeur à une variable, vous remplacez la valeur qu'elle contenait avant (on dit qu'on écrase la valeur précédente). La valeur précédente disparaît donc purement et simplement.

```
var solution = 25;  
solution = 13;
```

La variable `solution` contient la valeur `13` après ces deux lignes.

L'inconvénient de déclarer une variable avec le mot-clé `var`, c'est que cette variable devient utilisable partout dans le programme, ou partout dans la fonction dans laquelle elle est déclarée (nous parlerons des portées plus tard). C'est bien, direz-vous ! Oui et non. Parfois, on aimerait bien que la variable reste très locale à notre code, par exemple une variable de boucle n'a aucune raison d'être utilisable en dehors de la boucle (nous verrons ces notions plus loin, ne vous inquiétez pas). ES6 a introduit les deux mots-clés suivants :

- `let` : déclare une variable locale au bloc dans lequel elle est déclarée. Cette variable disparaît donc de la mémoire dès que le bloc est fermé.
- `const` : tout comme `let`, il déclare une variable locale, mais celle-ci est en lecture seule. C'est-à-dire qu'on l'affecte lorsqu'on la déclare, et qu'on ne peut plus la modifier par la suite. Ce qui en fait une constante, d'où le mot-clé. C'est comme si la boîte qui stocke l'information en mémoire était scellée, mais transparente : on peut voir le contenu, mais pas le changer.

2.1.2 Déclaration multiple

On peut aussi déclarer (et affecter) plusieurs variables en même temps en les séparant par des virgules :

```
let solution = 25,  
    recherche = 20,  
    test,  
    chaine;
```

Ici nous déclarons 4 variables différentes dont les deux premières sont directement affectées. Vous remarquerez qu'on peut aller à la ligne sans que ça gêne JavaScript... tant qu'il y a un point-virgule à la fin de l'instruction !

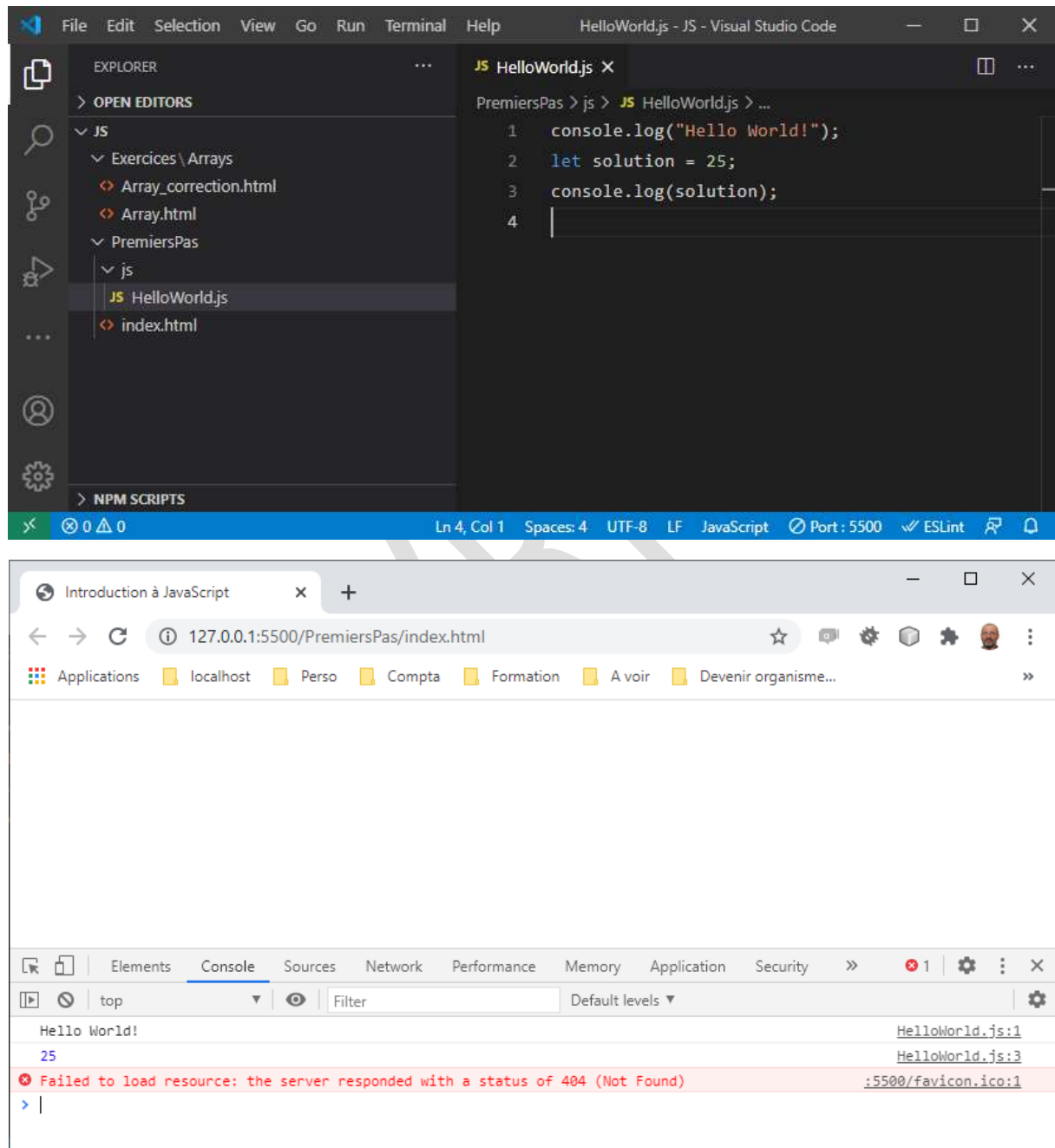
2.1.3 Lecture de variable

C'est bien gentil d'avoir stocké de l'information, mais ça serait sympa de pouvoir utiliser ces données. On aimerait donc pouvoir ouvrir la boîte et regarder ce qu'il y a dedans. Ça tombe bien, c'est fait pour !

En dehors d'une affectation, il suffit de donner l'identificateur de la variable et l'interpréteur le remplacera par sa valeur. Ainsi, si on veut afficher le contenu d'une variable dans la console (avec le traitement `log` déjà vu), on fera :

```
let solution = 25;
console.log(solution);
```

Si on ajoute ce code dans notre fichier `HelloWorld.js` et qu'on rafraîchit la page web, on doit obtenir :



Que se passe-t-il si on essaye de lire une variable déclarée mais pas encore affectée ? Essayez, vous obtiendrez la valeur `undefined`.

Que se passe-t-il si on essaye de lire une variable non déclarée ? Essayez, vous obtiendrez une erreur `ReferenceError : <variable> is not defined`.

2.1.4 Le nommage des variables

Avant toute chose, il est important de savoir que JavaScript est sensible à la casse (majuscule-minuscule), autrement dit les variables suivantes sont toutes différentes :

```
mavARIABLE
MaVariable
maVariable
MAVARIABLE
MavARIABLE
```

Attention donc à ne pas changer subitement d'identificateur dans votre code, vous allez avoir des erreurs partout.

Voici les règles obligatoires :

- Ne peuvent être un des mots réservés de JavaScript
- Doivent contenir des caractères alphanumériques
- Ne doivent pas commencer par un chiffre
- Ne contiennent pas de caractères de contrôle
- Ne contiennent pas d'espace ni de ponctuation (à part *dollar* `$` ou *underscore* `_`)
- Ne doivent pas faire suivre un caractère accentué par un chiffre

Voici quelques règles d'or :

- Choisir des noms significatifs. En effet, `perimetre` est plus simple à comprendre que `var1`.
- Ne pas utiliser de caractères ésotériques (accent, cédille, tilde et autres joyeusetés). JavaScript les accepte, mais tous les claviers n'y ont pas accès facilement (et pour un anglais, lire *périmètre* est pour le moins périlleux).
- Ne pas utiliser des noms trop long. Simplement pour avoir des lignes moins longues, et puis c'est long à taper... Quoique certains choisissent de mettre des noms à rallonge et ne mettent jamais de commentaires. C'est un choix qui vous appartient.
- Utiliser une convention de nommage. La plus répandue est le *camel case* qui consiste à mettre une majuscule au début de chaque mot constituant l'identificateur de la variable. Par exemple, `ceciEstLeNomDeMaVariable`.
- Les identificateurs en majuscules sont réservés pour les constantes (les variables déclarées avec `const`).

Comme pour tout langage de programmation, certains mots sont interdits : ceux utilisés par le langage lui-même. En effet, si vous définissez une fonction qui a le même nom qu'une fonction du langage, comment l'interpréteur saura-t-il quelle fonction utiliser ? Voici la liste des mots réservés de JS :

await	break	case	catch	class	const	continue	debugger	default
delete	do	else	export	extend	finally	for	function	if
import	in	instanceof	new	return	super	switch	this	throw
try	typeof	var	void	while	with	yield		

Sont réservés aussi pour le futur : `implements`, `package`, `protected`, `interface`, `private`, et `public`.

2.1.5 Exceptions de nommage

Je ne peux pas passer sous silence qu'il est tout à fait possible, en JavaScript, d'utiliser une variable sans l'avoir déclarée. Par exemple, le code suivant est parfaitement fonctionnel :

```
uneVariable = 27;
```

Même si `uneVariable` n'a jamais été déclarée par un :

```
var uneVariable;
```

Ceci reste cependant une très mauvaise pratique. D'une part parce que votre code devient illisible (tout programmeur va se demander d'où vient cette variable et chercher désespérément sa déclaration), d'autre part parce que cette variable existe bien « quelque part » et nous verrons plus tard où, ce qui revient somme toute à en faire la déclaration au bon endroit, ce qui est bien plus propre.

2.1.6 Variable globale

Utiliser une variable globale, c'est mal ! Éviter les variables globales est une bonne hygiène de programmation !

En *mode strict* (nous verrons plus tard ce que c'est) l'affectation sans déclaration préalable provoque une erreur `ReferenceError` :

```
"use strict"; maVar = 1; // Erreur !
```

Hors mode strict, il peut sembler possible d'affecter une variable sans l'avoir déclarée, et ce n'importe où dans le code :

```
maVar = 1; // Pourtant maVar n'a pas été déclarée avant
```

Si JavaScript s'exécute dans le contexte d'un navigateur, en fait `maVar` ne sera pas à proprement parler une variable. JavaScript ajoute simplement une propriété `maVar` à l'objet global `window`.

```
console.log( maVar in window ); // true en environnement navigateur
```

Comme l'objet global `window` est accessible partout et que ses propriétés sont scrutées quand JavaScript rencontre un identifiant inconnu, `maVar` s'utilisera alors comme une sorte de « variable globale » accessible dans tout le code.

C'est en fait aussi ce qui se passe quand on déclare une variable avec `var` en dehors de toute fonction.

2.2 Les types de variable

JS propose 8 types de données, 7 dits *primitifs* et un dit *complexe* en référence à la terminologie Java :

- Les nombres (type *number*)
- Les chaînes de caractères (type *string*)
- Les booléens (type *boolean*)
- La valeur nulle (type *null*)
- La valeur indéfinie (type *undefined*)
- Les symboles (type *symbol*) depuis ES6 seulement (2015)
- Les grands entiers (type *bigint*) depuis ES11 seulement (2020)
- Les objets (type *object*)

Sachez toutefois que les types primitifs peuvent être transformés en type complexe grâce aux wrappers qui sont des classes (nous y reviendrons) qui contiennent la valeur primitive et apportent des fonctionnalités supplémentaires.

Il est possible de connaître le type d'un variable avec l'opérateur `typeof`. Une exception toutefois avec le type `null`, car `typeof` renvoie alors le type `object`.

type	typeof	wrapper
Nombre	number	Number
Chaîne de caractères	string	String
Booléen	boolean	Boolean
Valeur nulle	object (!)	N/A
Valeur indéfinie	undefined	N/A
Symbole (valeur unique et immuable)	symbol	Symbol
Entiers sans limite	bigint	BigInt
Objet	object	Object

2.2.1 Les nombres

Dans la plupart des langage il existe des types distincts pour les entiers et les décimaux. En JS, non : c'est le même type `number`. Ils sont systématiquement représentés en mémoire au format double précision sur 64 bits de la norme IEEE 754-2008.

On peut ainsi représenter des nombres de valeur absolue comprise entre $2.2e-308$ et $1.8e+308$ environ, et on a accès à tous les nombres entiers entre -2^{53} et $2^{53}-1$. Au-delà de ces valeurs, on a accès à certains entiers mais pas tous... De l'intérêt du wrapper `Number` qui possède la propriété `MAX_SAFE_INTEGER`.

Exercice : dans la console regardez la valeur de `2**53`, `2**53+1` et `2**53+2`

NB : depuis ES11, le type `bigint` permet de gérer tous les entiers (cf infra §2.2.1.4).

2.2.1.1 Écriture littérale

On peut écrire un nombre en base 10, 16, 8 ou 2.

Les nombres en base 10 s'écrivent de la façon la plus habituelle avec un point décimal anglais : 1.3, +1.1, -0.5, -1 ou en écriture scientifique : $1.12E-3$ (qui signifie 1.12×10^{-3}).

Les nombres en base 2 doivent être précédés d'un `0b` : `0b1012` = `1010`.

Les nombres en base 16 doivent être précédés d'un `0x` : `0x1816` = `2410`.

Les nombres en base 8 doivent être précédés d'un `0o` : `0o178` = `1510`.

Attention : les navigateurs acceptent également la notation octale en précédant le nombre d'un simple `0` : si les chiffres sont interprétables comme de l'octal (entre 0 et 7), ce nombre sera octal, sinon décimal. Cette notation est à proscrire car sujette à mauvaise interprétation. D'ailleurs, le mode strict l'interdit.

Exemple :

```
console.log(08); // interprété comme décimal, et vaut 8
console.log(010); // interprété comme octal et vaut 8
```

```
"use strict"; console.log(010); // Erreur
```

2.2.1.2 Nombres particuliers

Des valeurs particulières existent en tant que nombre :

- `NaN` qui signifie *Not a Number*, est donc un nombre non représentable (comme $0/0$ par exemple, ou $\log(-1)$)
- `+Infinity` et `-Infinity` représentent les infinis, qui sont plus grand (ou plus petit) que tout autre nombre.

Ces nombres ont leur propre algèbre :

- Toute opération mathématique impliquant `NaN` aura pour résultat `NaN`
- Toute opération mathématique impliquant l'infini aura pour résultat l'infini, sauf dans le cas d'une valeur indéterminée qui vaudra `NaN` (comme `+Infinity-Infinity`)

Il est possible de tester ces valeurs avec les fonctions `isFinite()` et `isNaN()` qui retournent un booléen :

```
isFinite(Infinity); // false
isNaN(NaN); // true
```

2.2.1.3 Remarques sur les nombres flottants

La représentation de nombres avec un système de mantisse et d'exposant¹ impose malheureusement des contraintes. En particulier, certaines valeurs ne sont pas exprimables. C'est la même chose en base 10 : vous ne savez pas écrire $1/3$ en décimal. Vous ne pouvez en avoir qu'une valeur approchée, certes précise, mais toujours approchée. En Javascript, c'est pareil. La représentation des nombres se fait en base 2 dans la mémoire, et certaines valeurs sont forcément approchées. Il faut donc bien penser que faire une égalité de valeurs est forcément dangereux.

Exercice : quel est la valeur du test `0.1 + 0.2 == 0.3` ?

Il faut en particulier faire attention à deux choses :

- L'*underflow* qui est l'impossibilité de représenter des nombres proches de 0 ($1E-400$)
- L'*overflow* qui est l'impossibilité de représenter des nombres au-delà d'un certain seuil (`Number.MAX_VALUE` ou `Number.MIN_VALUE`)
- L'*epsilon* qui est le plus petit intervalle matérialisable (`Number.EPSILON`)

Pour comparer deux nombres en JS, si on flirte avec ces limites, il faut utiliser un code du genre :

```
x = 0.2;
y = 0.3;
equal = (Math.abs(x - y) < Number.EPSILON);
```

2.2.1.4 Les grands entiers

Depuis 2020 (ES11), un nouveau type primitif est arrivé : le type `bigint` qui permet de gérer des nombres entiers aussi grands qu'on le veut. Pour déclarer un tel entier, il suffit de le suffixer par la lettre `n` :

```
let a = 1n ; typeof a; // bigint
```

¹ Voir à ce sujet ces deux excellents sites (en anglais) : <https://floating-point-gui.de/> ainsi que <https://medium.com/angular-in-depth/the-mechanics-behind-exponent-bias-in-floating-point-9b3185083528>

Quelque chose de très important : on ne peut pas mixer des `bigint` avec des `number` dans un calcul :

```
let a = 1n; // un bigint
let b = 1;  // un entier (number)
console.log(a+b); // erreur
```

Il est nécessaire de transtyper `b` en `bigint` auparavant en utilisant le wrapper :

```
console.log(a+BigInt(b));
2n
```

2.2.1.5 Le wrapper `Number`

La fonction `Number()` retourne un `number` à partir d'une expression :

```
Number("17") // Retourne le number 17 (entier)
Number("3.14") // Retourne le number 3.14 (flottant)
```

Cet objet apporte de plus de nombreuses informations complémentaires et fort utiles :

```
Number.MIN_SAFE_INTEGER // Plus petit entier représentable sans risque
Number.MAX_SAFE_INTEGER // Plus grand entier représentable sans risque,
Number.MAX_VALUE        // Plus grande valeur représentable
Number.MIN_VALUE        // Plus petite valeur positive représentable
Number.NaN              // La valeur NaN
Number.EPSILON          // Plus petit intervalle de valeurs représentable
```

Ainsi que des fonctionnalités de transtypage :

```
Number.parseInt("17") // Retourne le number 17 (entier)
Number.parseFloat("3.14") // Retourne le number 3.14 (flottant)
Number.isFinite(17) // Retourne le booléen true
Number.isNaN(NaN) // Retourne le booléen true
Number.isInteger(3.12) // Retourne le booléen false
Number.isSafeInteger(2**53+5) // Retourne le booléen false
```

2.2.2 Les chaînes de caractères

Une chaîne de caractères est une succession d'au plus $2^{53}-1$ caractères représentés par des entiers non signé sur 16 bits (Unicode UTF-16).

Une chaîne de caractères est non modifiable ou *immuable* (*immutable* en anglais). Dès qu'on croit modifier une chaîne, en fait on en crée une nouvelle sur la base de la première.

2.2.2.1 Écriture littérale

Un littéral `string` peut s'écrire indifféremment avec les délimiteurs `'` ou `"`.

```
'Bonjour'
"Bonjour"
"Bonjour" // Incorrect
```

Le caractère délimiteur ne peut pas faire partie de la chaîne sans être préfixé par le caractère `\`. Ce caractère `\` ayant un sens particulier, il doit lui-même être préfixé si on veut l'écrire dans la chaîne :

```
'Le "mot"' // Correct
"Aujourd'hui" // Correct
'Aujourd'hui' // Incorrect
"Cette table est "maganée" comme on dit ici" // Incorrect
```



```
"Cette table est \"maganée\" comme on dit ici" // Correct
'Un antislash \' \' // Correct
```

De la même manière, certains caractères (non imprimables) peuvent être écrits :

saut de ligne	<code>\n</code>	tabulation	<code>\t</code>	tabulation verticale	<code>\v</code>	retour arrière	<code>\b</code>
retour chariot	<code>\r</code>	saut de page	<code>\f</code>	caractère de code 0	<code>\0</code>		

2.2.2.2 Concaténation

L'opérateur `+` fournit une nouvelle `string` cumulant les caractères des deux opérandes :

```
'Bonjour ' + 'le monde.' // Nouvelle string 'Bonjour le monde.'
```

2.2.2.3 Le wrapper String

La fonction `String()` retourne une `string` à partir d'une expression :

```
String(123) // "123"
String(3 * 2) // "6"
String(NaN) // "NaN"
String(1/0) // "Infinity"
String(1==1) // "true"
String(null) // "null"
```

L'objet wrapper `String()` fournit de nombreuses propriétés pour manipuler les chaînes. On peut utiliser ces propriétés sur une donnée de type `string`.

La propriété `length` donne le nombre de caractères :

```
'abcdef'.length // 6
```

La méthode `charAt()` retourne le caractère présent à la position donnée (à partir de 0) :

```
'yop'.charAt(0) // 'y'
'yop'.charAt(2) // 'p'
```

NB : depuis ES6, on peut aussi lire (mais pas modifier) chaque caractère d'une chaîne par son indice avec la notation tableau `[]`. L'indexation commence toujours à 0.

```
'yop'[1] == 'yop'.charAt(1); // true
```

Lire [MDN : String](#). Vous y trouverez de nombreuses propriétés utiles de l'objet `String`.

2.2.2.4 Gabarits

Depuis ES6, on peut créer des *gabarits* (*Template Strings*) avec ```. Dans un gabarit, l'expression contenue dans `${ }` est évaluée et les caractères saut de ligne sont possibles.

```
let nbMoto = 3, nbVoiture = 10;
console.log(`Quantité
Moto : ${nbMoto + 1}
Voiture : ${ nbVoiture }`);

Quantité
Moto : 4
Voiture : 10
```

Les gabarits sont riches en fonctionnalités, lecture conseillée [MDN : littéraux de gabarit](#).

2.2.3 Les booléens

Les variables booléennes sont des variables qui ne contiennent qu'une valeur logique : Vrai ou Faux (plus précisément en JavaScript : `true` ou `false`). Leur nom vient d'un mathématicien et philosophe anglais du XIX^{ème} siècle, *George Boole*, qui crée entre 1844 et 1854 une algèbre binaire à laquelle il donne son nom : l'algèbre booléenne. En 1938, *Claude Shannon* reprendra le travail de Boole et fondera avec quelques pairs la théorie de l'information que tous les ordinateurs utilisent.

L'objet wrapper `Boolean()` permet d'obtenir un `boolean` à partir d'une expression. La règle appliquée est la suivante : les valeurs `0`, `""` (chaîne vide), `NaN`, `null` et `undefined` donneront `false`. Toute autre valeur donnera `true`.

```
Boolean(123) // true
Boolean(5 - 5) // false
Boolean("Bonjour") // true
```

2.2.4 Type indéfini

Ce type n'admet qu'une unique valeur `undefined`.

Par exemple, une variable non encore affectée aura pour valeur `undefined`; une fonction sans `return` retourne `undefined`.

2.2.5 Type symbole

Ce type est apparu avec ES6. Un symbole est une donnée *unique et immuable*.

L'intérêt ? C'est un moyen plus robuste que les chaînes pour représenter des identificateurs (comme clé d'un objet, comme élément d'une énumération, etc...) :

```
const ANIMAL_CHIEN = Symbol();
const ANIMAL_CHAT = Symbol();
switch (animal) {
  case ANIMAL_CHIEN: // ici le code dans le cas du chien
  case ANIMAL_CHAT: // ici le code dans le cas du chat
}
```

On peut mettre une définition du symbole en argument de `Symbol()`, mais il est important de bien comprendre que ce n'est qu'une information sur le symbole, pas son nom. Ainsi :

```
Symbol('toto') == Symbol('toto') // retourne false
```

2.2.6 La valeur nulle

Ce type n'admet qu'une unique valeur `null`, généralement employée pour signifier l'absence d'objet (à ne pas confondre avec la valeur numérique nulle `0`).

```
let monObjet = null; // variable monObjet est affectée de la valeur null
```

2.2.7 Les objets

Nous verrons plus loin les objets plus en détail, mais on peut déjà les introduire rapidement.

2.2.7.1 Propriétés

Un objet est un ensemble de zéro, une ou plusieurs propriétés. Une propriété est une association entre un *nom* et une *valeur*. Les valeurs peuvent être de tout type, primitif ou objet.

NB : comme une fonction est un objet en JS, une propriété peut donc être une fonction. Dans ce cas on l'appellera une méthode.

Pour accéder (en lecture / écriture) à la propriété d'un objet, JavaScript fournit deux notations : le point `.` et les crochets `[]`.

2.2.7.2 Intérêt des objets

Supposons que dans notre programme, on ait besoin de manipuler des voitures. Le programme n'a que faire d'une voiture du monde réel. Il a besoin d'une représentation simplifiée (une modélisation) du concept de voiture. Par exemple notre programme doit manipuler les données suivantes : la marque, le modèle et la cylindrée. Un objet qui contiendrait les propriétés marque, modèle, et cylindrée serait tout indiqué pour manipuler facilement une « voiture » dans notre programme.

Voyons cela ci-dessous :

```
let o = new Object();           // ❶
o.marque = 'Bugatti';          // ❷
o.modèle = 'Type 2';           // ❸
o.cylindrée = 4;                // ❹
console.log(`Marque: ${o.marque}, modèle: ${o.modèle}, cylindrée:
${o.cylindrée}`);              // ❺
```

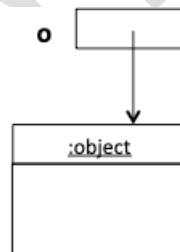
Marque: Bugatti, modèle: Type 2, cylindrée: 4

En ❶, on crée un objet avec le mot-clé `new` et une fonction qui agit comme constructeur d'objet. Ici on utilise la fonction `Object()` fournie par JavaScript qui permet justement de créer un objet générique.

Au niveau du vocabulaire, on dit que l'objet ainsi créé est un objet `Object` ou est un objet de type `Object`, ou encore simplement que c'est un `Object`.

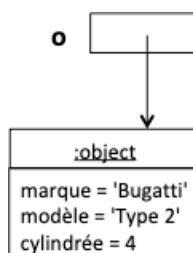
On affecte cet objet à la variable `o` pour pouvoir le manipuler ensuite.

Voici comment on pourrait se représenter alors la situation dans la mémoire de la machine :



TRÈS IMPORTANT ! En mémoire, la variable `o` contient une référence vers l'objet. La variable ne contient pas les données de l'objet à proprement parler ! CEPENDANT, par abus de langage, on parlera de « l'objet `o` ».

En ❷, ❸ et ❹ on ajoute des propriétés à l'objet :



Enfin en ❺ on affiche les propriétés sur la console.

C'est une particularité de JavaScript (par rapport à des langages comme Java) que de pouvoir ajouter dynamiquement des propriétés à un objet après sa création.

On modifie la valeur d'une propriété de la même façon :

```
o.modèle = 'Type 3';           // Propriété modèle mise à jour
o.année = 1903;                // Propriété année mise à jour
```

Dans une expression, une propriété qui n'a jamais été ajoutée vaut `undefined` :

```
console.log(o.année);          // Affichage de undefined
o.année = 1900;                // Ajout de la propriété
console.log(o.année);          // Affichage de 1900
```

On peut aussi parfois vouloir retirer (complètement) une propriété d'un objet (avec `delete`) :

```
delete o.année;                // Retire la propriété de l'objet
console.log(o.année);          // Affiche undefined
```

La présence d'une propriété se teste avec l'opérateur `in` :

```
console.log('marque' in o);     // true
console.log('couleur' in o);    // false
```

Comme dit plus haut, il existe une seconde syntaxe `[...]` pour manipuler les propriétés d'un objet.

Le nom de la propriété est alors fourni entre les crochets sous forme de `string` :

```
o['année'] = 1903;
console.log(o['année']);        // 1903
```

2.2.7.3 Objets littéraux

Il existe une notation spécifique à JavaScript pour écrire les objets de façon littérale. Ceci permet d'écrire une expression évaluée à un objet sans recourir à `new`.

Un littéral objet s'écrit entre accolades `{}` et `}`, et :

- On définit des propriétés sous la forme `nom : valeur`.
- Chaque définition de propriété est séparée des autres par une virgule `,`.

```
{ }                             // un objet sans propriété
{ nom : 'Paul', âge : 15 }      // un objet avec 2 propriétés : nom et âge
```

Cette notation permet d'écrire très facilement un littéral objet complexe. Dans l'exemple qui suit on affecte à une variable `o` un objet créé littéralement :

```
let o = {
  prénom : 'Marcel',
  âge : 12,
  parents : [
    { prénom : 'Jacques', âge : 37 },
    { prénom : 'Lea', âge : 35 }
  ],
  boursier : true
}
```

Certaines syntaxes de l'exemple ci-dessus vont être décrites dans les chapitres qui suivent donc ce n'est pas grave si tout n'est pas limpide. La notation représente ici un objet composé de 5 propriétés :

- prénom (contenant la string Marcel)
- âge (contenant le number 12)
- parents (contenant un tableau de 2 éléments). Chaque élément est lui-même un objet avec 2 propriétés prénom et âge
- boursier (contenant la valeur booléenne true).

2.2.7.4 Objets globaux

Ce sont des objets natifs et donc immédiatement disponibles.

Ne pas confondre ce terme avec l'*objet global*, qui est un objet très précis dont l'existence est liée à l'environnement (`window` en environnement navigateur par exemple, `global` dans *nodeJS*).

On y trouve des objets utiles comme `Number`, `Math`, `Object`, `Array`, `String`, ... Suivant les cas, on aura à créer ces types d'objets, parfois on utilisera juste les propriétés rendues disponibles :

Exemples utiles :

```
let noel = new Date(2018, 12, 25); // construction d'un objet Date
let alea = Math.random();          // nombre pseudo-aléatoire 0 ≤ alea < 1
x = Math.round(3.14);              // Arrondi à l'entier le plus proche (3)
```

Pour aller plus loin

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Date

2.2.7.5 Affectation d'un objet

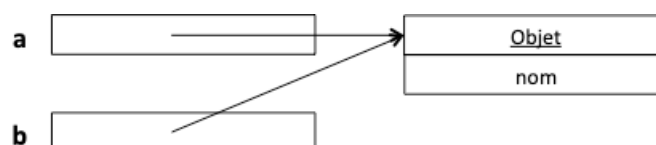
Quand on affecte un objet à une variable, c'est la référence de l'objet qui est stockée dans la variable. Ceci a de nombreuses implications :

```
const a = { <propriétés> };
```



En particulier copier la valeur de la variable `a` dans la variable `b` ne crée pas de nouvel objet :

```
const a = { nom : 'John' };
const b = a;
b.nom = 'Martin';
a.nom === b.nom           // true
a === b                   // true
a === b                   // true
console.log(a.nom);       // Martin
```



Remarquons également au passage que `a` et `b` ont été déclarés en tant que constante, et pourtant on a pu modifier l'objet. C'est légitime puisque `a` comme `b` sont certes des constantes, mais c'est leur

valeur qui ne peut plus changer une fois affectée. Or, cette valeur, c'est la référence à l'objet, pas l'objet lui-même. On peut donc effectivement modifier l'objet référencé par `a` (ou `b`). En revanche, on ne pourra plus réaffecter un autre objet (ou un autre type d'ailleurs) à `a` ni à `b`.

2.3 Les opérateurs

2.3.1 Liste

Les *expressions* JavaScript sont construites à partir des *variables*, des *valeurs littérales* et des combinaisons qu'on peut en faire avec les opérateurs que voici :

Précédence	Type d'opérateur	Associativité	Opérateurs individuels
0	Groupement	Non applicable	(...)
1	Accès à un membre	Gauche à droite
	Accès à un membre calculé	Gauche à droite	... [...]
	new (avec une liste d'arguments)	Non applicable	new ... (...)
	Appel de fonction	Gauche à droite	... (...)
	Chaînage optionnel	Gauche à droite	?.
2	new (sans liste d'arguments)	Droite à gauche	new ...
3	Incrémementation suffixe	Non applicable	... ++
	Décrémementation suffixe	Non applicable	... --
4	NON logique	Droite à gauche	! ...
	NON binaire	Droite à gauche	~ ...
	Plus unaire	Droite à gauche	+ ...
	Négation unaire	Droite à gauche	- ...
	Incrémementation préfixe	Droite à gauche	++ ...
	Décrémementation préfixe	Droite à gauche	-- ...
	typeof	Droite à gauche	typeof ...
	void	Droite à gauche	void ...
	delete	Droite à gauche	delete ...
	await	Droite à gauche	await ...
5	Exponentiation	Droite à gauche	... ** ...
	Multiplication	Gauche à droite	... * ...
	Division	Gauche à droite	... / ...
	Reste	Gauche à droite	... % ...
6	Addition	Gauche à droite	... + ...
	Soustraction	Gauche à droite	... - ...
7	Décalage binaire à gauche	Gauche à droite	... << ...
	Décalage binaire à droite	Gauche à droite	... >> ...
	Décalage binaire à droite non-signé	Gauche à droite	... >>> ...
8	Inférieur strict	Gauche à droite	... < ...
	Inférieur ou égal	Gauche à droite	... <= ...
	Supérieur strict	Gauche à droite	... > ...
	Supérieur ou égal	Gauche à droite	... >= ...
	in	Gauche à droite	... in ...
	instanceof	Gauche à droite	... instanceof ...

9	Égalité faible	Gauche à droite	... == ...
	Inégalité faible	Gauche à droite	... != ...
	Égalité stricte	Gauche à droite	... === ...
	Inégalité stricte	Gauche à droite	... !== ...
10	ET binaire	Gauche à droite	... & ...
11	OU exclusif (XOR) binaire	Gauche à droite	... ^ ...
12	OU binaire	Gauche à droite
13	ET logique	Gauche à droite	... && ...
14	OU logique	Gauche à droite
15	Opérateur conditionnel ternaire	Droite à gauche	... ? ... : ...
16	Affectation	Droite à gauche	... = ...
			... += ...
			... -= ...
			... *= ...
			... /= ...
			... **= ...
			... %= ...
			... <=<= ...
			... >>= ...
			... >>>= ...
			... &= ...
			... ^= ...
			... = ...
17	yield	Droite à gauche	yield ...
	yield*	Droite à gauche	yield* ...
18	Décomposition	Non applicable
19	Virgule	Gauche à droite	... , ...

La *précédence* indique l'ordre dans lequel les opérateurs sont appliqués : la précédence 0 aura donc priorité sur la précédence 19.

Exemple : dans l'expression $1+2**2$ se trouvent 2 opérateurs : l'addition $+$ et l'exponentiation $**$. L'opérateur d'exponentiation $**$ est d'un niveau de priorité plus élevé (précédence 5) que l'opérateur d'addition $+$ (précédence 16). Donc l'opération $2**2$ est évaluée en premier et vaut 4. L'expression à évaluer devient alors $1+4$ ce qui est évalué à 5, valeur finale.

Connaître les priorités c'est bien, mettre des parenthèses c'est mieux ! Remarquez que les parenthèses de groupement ont la précédence 0, donc seront toujours évaluées en premier. Sans compter que ça simplifie la lecture...

L'associativité indique dans quel sens seront évaluées des opérations de même précédence.

Exemple : dans l'expression $2**3**2$, les 2 opérateurs ont la même précédence (5). Cependant le sens d'associativité de l'opérateur $**$ est de droite à gauche, donc la première sous-expression à être évaluée est $3**2$ ce qui donne 9. Enfin l'opération $2**9$ est évaluée ce qui donne 512 comme valeur de l'expression initiale.

2.3.2 Transtypage automatique des opérandes

Les opérateurs de JavaScript acceptent souvent des opérandes d'un type autre que celui attendu. Dans ce cas, JavaScript applique une règle de conversion (on parle de transtypage).

ATTENTION ! Le résultat de cette conversion peut être inattendu, surtout quand on utilise d'autres langages (qui appliquent d'autres règles). Je conseille fortement d'éviter les conversions implicites quand c'est possible, et au pire de documenter les non triviales. Surtout quand on débute...

```
"2" - 1           // 1 car "2" est converti en 2
"2" * "1"         // 2 car "2" est converti en 2 et "1" en 1
"2"-true          // 1 car true est converti en 1 (false en 0)
```

2.3.3 Opérateur +

On peut l'utiliser naturellement avec deux nombres. On sait qu'on peut aussi l'utiliser avec deux string, c'est une concaténation :

```
1 + 2             // 3
"ab" + "cd"       // "abcd"
```

Mais on peut l'utiliser avec beaucoup d'autres choses, et selon le type des opérandes, cet opérateur va appliquer 3 règles (dans l'ordre !) :

Règle r1) Si un opérande est un objet, celui-ci sera converti en un type primitif (si `valueOf()` le permet) ou en une string (avec `toString()`).

```
"Le " + Date()    // "Le " + Date().toString() (r1)
```

Règle r2) Si après conversion un opérande est de type string, le second est converti en type string ce qui donne lieu à une concaténation :

```
"2" + null // "2" + "null" (r2) donc "2null"
2 + "1" // "2" + "1" (r2) donc "21"
```

Règle r3) Sinon les deux opérandes sont convertis en number, et le résultat est la somme.

```
3 + true // 3 + 1 (r3) donc 4
4 + null // 4 + 0 (r3) donc 4
```

Ces règles s'appliquent en cascade :

```
[1, 2, 3] + 4 // [1,2,3].toString() + 4 (r1)
               // donc "1,2,3"+"4" (r2) donc "1,2,34"
1 + {}        // 1 + "[object Object]" (r1)
               // donc "1" + "[object Object]" (r2)
               // donc "1[object Object]"
```

Attention à garder la logique de votre code facilement compréhensible. Ne pas en abuser et au besoin commentez.

2.3.4 Opérateurs logiques

JavaScript ne limite pas les opérateurs logiques (`!`, `&&`, `||`) aux simples valeurs booléennes.

Évidemment pour les valeurs booléennes, les tables de vérité classiques sont respectées. Mais si on utilise d'autres types, voici le fonctionnement standard de ces opérateurs :

- Fonctionnement du ET logique (`&&`) : `a && b` vaut `a` si `a` peut être converti en `false`, sinon `b`.

- Fonctionnement du OU logique (||) : a || b vaut a si a peut être converti en true, sinon b.
- Fonctionnement du NON logique (!) : !a vaut false si a peut être converti en true, sinon true.

La tentative de conversion d'un type primitif en boolean suit la règle : null, 0, une string vide, undefined sont convertis en false. Le reste est converti en true.

Par exemple :

```

true && true           // true
10 && 5                // 10 convertissable en true, donc 5
0 && 2                 // 0 convertissable en false, donc 0
false || 'toto'        // 'toto'
(10 && 5) === (3 && 2)  // équivalent à 5 === 2, donc false
!'toto'                // équivalent à !true, donc false

```

Notez que null comme undefined sont convertis en false, ce qui est un peu gênant, puisqu'en fait ces deux valeurs sont vides. Un nouvel opérateur existe depuis ES11 (2020) : le nullish coalescing ??.

- Fonctionnement du nullish coalescing logique (??) : a ?? b vaut a si a n'est pas vide (null ou undefined) sinon b.

Exemples :

```

console.log("superToto" ?? "Toto"); // "superToto"
console.log(false ?? "Toto");       // false
console.log(undefined ?? "Toto");   // "Toto"
console.log(null ?? "Toto");        // "Toto"

```

2.3.5 Opérateurs de comparaison

2.3.5.1 Égalité faible ou forte

Il y a une différence importante entre l'opérateur d'égalité faible == et l'opérateur d'égalité stricte ===.

Dans une comparaison d'égalité faible, JavaScript essaie d'abord de convertir les opérandes en un même type avant d'effectuer la comparaison. L'égalité stricte ne sera vraie que si les opérandes sont de même type ET égaux.

Comme il n'y a pas de conversion, quand le choix est possible, plutôt choisir l'opérateur d'égalité stricte ===.

```

let a = 1, b = 1.0;
a == b           // true
a == true        // true (après conversion)
a === b          // true car 1 et 1.0 sont des number.
a === true       // false car 1 et true ne sont pas de même type.

```

2.3.5.2 Règle de conversion appliquée dans une Egalite faible entre opérandes de types différents

La table ci-dessous résume les règles de conversion appliquées dans le cas d'une opération égalité faible A==B en fonction des types de A et B :

A / B	Undefined	Null	Number	String	Boolean	Object
Undefined	true	true	false	false	false	false
Null	true	true	false	false	false	false
Number	false	false	A === B	A === ToNumber(B)	A === ToNumber(B)	A == ToPrimitive(B)

String	false	false	ToNumber(A) === B	A === B	ToNumber(A) === ToNumber(B)	A == ToPrimitive(B)
Boolean	false	false	ToNumber(A) === B	ToNumber(A) === ToNumber(B)	A === B	false
Object	false	false	ToPrimitive(A) == B	ToPrimitive(A) == B	ToPrimitive(A) == ToNumber(B)	A === B

Il est bien sûr hors de question d'apprendre ce tableau par cœur, en revanche il faut savoir qu'il existe. On le retrouve sur MDN : les tests d'égalité. Ici encore, évitez dans votre code ce qui risque d'être mal compris par un autre développeur, par exemple en commentant utilement.

`toNumber()` dans ce tableau signifie que JavaScript va essayer de convertir en `number` avant la comparaison et `toPrimitive()` signifie qu'il va essayer de convertir en une valeur de type primitif (avec `toString()`, ou `valueOf()`).

Par exemple, si JavaScript rencontre l'expression `'1' == true`, il considère la situation `<string>==<boolean>`. L'évaluation sera donc d'après la table `toNumber('1') == toNumber(true)`, c'est-à-dire `1 == 1`, et donc `true`.

```
1 == true    // conversion (1 === toNumber(true)) donc 1 === 1 donc true
2 == true    // conversion (2 === toNumber(true)) donc 2 === 1 donc false
1 == {}      // conversion en 1 === toPrimitive({})
              // donc 1 == "[object Object]"
              // conversion en 1 == toNumber("[object Object]")
              // donc 1 == NaN donc false
```

2.3.5.3 Égalité faible vs affectation

Attention à l'erreur classique qui consiste à utiliser l'opérateur d'affectation `=` à la place de l'opérateur d'égalité faible `==`.

C'est syntaxiquement correct puisqu'une affectation est aussi une expression (qui vaut la valeur affectée), mais probablement pas ce qu'on voulait écrire. N'a pas encore vraiment codé celui qui n'a pas fait cette erreur au moins une fois...

```
let a = 2;
if (a = 3) { <bloc> } // (a=3) vaut 3 qui est convertissable en true
                      // donc on entre dans le bloc et a vaut maintenant 3

let a = 2;
if (a==3) { <bloc> } // 2 ≠ 3, on n'entre pas dans le bloc, a est inchangé
```

2.3.5.4 Comparaison de string

Les chaînes sont comparées caractère par caractère dans l'ordre lexicographique de l'Unicode.

```
'a' < 'b'           // true
'' < 'a'             // true
'b' > 'abbbb'        // true
'A' < 'a'            // true car code(A) = 65 < code(a) = 97
'ab' < 'ac'          // true
```

2.3.5.5 Opérateurs d'incrément et de décrémentation

Attention à la position de l'opérande. Ne confondez pas valeur de la variable et valeur de l'expression.

```
++a           // a est incrémenté et l'expression vaut a+1
a++           // a est incrémenté et l'expression vaut a
```

N'utilisez pas ces opérateurs quand ils risquent de prêter à confusion.

2.3.5.6 Opérateur virgule

L'opérateur virgule `,` évalue chacun de ses opérandes (de gauche à droite). L'expression a pour valeur le dernier des opérandes.

```
1+2, 5 // 1+2 est évalué à 3, 5 est évalué à 5. L'expression vaut 5.
```

Il permet parfois de simplifier l'écriture mais n'est pas indispensable.

2.3.5.7 Opérateur typeof

L'opérateur `typeof` retourne le type d'une expression sous forme de chaîne de caractères :

```
typeof 3.14 // "number"
typeof true // "boolean"
typeof "ab" // "string"
typeof {} // "object"
```

Notons les cas particuliers :

```
typeof null // "object" et non pas "null"
typeof function // "function" et non pas "object"
```

2.3.5.8 Opérateur conditionnel ternaire

Il renvoie une valeur ou une autre en fonction d'une condition (une expression booléenne). L'expression `A ? B : C` vaut `B` si `A` est `true`, ou `C` si `A` est `false`.

```
'truc' + (qté > 1 ? 's' : '') // affiche 'truc' si la variable qté >1,
                             //sinon 'trucs'
```

2.3.5.9 Opérateurs bit à bit

Ils opèrent une comparaison bit à bit. Rarement utilisés sauf si on sait faire !

Attention à ne pas confondre les opérateurs logiques et les opérateurs de comparaison bit à bit.

```
10 && 5 // ET logique, vaut 5
10 & 5 // ET bit à bit, équivalent à 0b1010 & 0b0101, donc 0b0000 i.e. 0
10 | 5 // OU bit à bit, équivalent à 0b1010 | 0b0101, donc 0b1111 i.e. 15
```

2.3.5.10 Opérateur de chaînage optionnel

Introduit par ES11 (2020) cet opérateur permet de se protéger contre les valeurs `null` ou `undefined` lors de tests. Lorsqu'on essaye d'accéder à un objet qui n'existe pas, ou à la propriété inexistante d'un objet, ça provoque une erreur :

```
const personne = {
  voiture: 'Bugatti',
  conjoint: {
    prénom: 'Mauricette'
  }
};
// erreur Cannot read property 'nom' of undefined
const enfant = personne.enfant.nom;
console.log(enfant);
// erreur personne.méthodeInexistante is not a function
console.log(personne.méthodeInexistante());
```

Il est donc nécessaire de faire un test d'existence avant de faire la première assignation, ou l'appel à la fonction inexistante :

```
const enfant = personne.enfant ? personne.enfant.nom : undefined;
```

Le problème est que c'est assez lourd à écrire. L'opérateur de chainage optionnel `?.` vient alléger l'écriture :

```
const enfant = personne.enfant?.nom;           // Pas d'erreur  
console.log(enfant);                           // affiche undefined  
console.log(personne.méthodeInexistante?.());  // affiche undefined
```

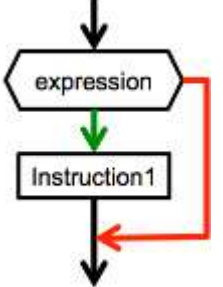
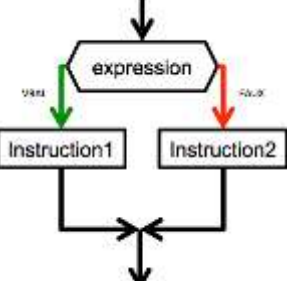
Notez l'écriture de l'appel à la fonction : l'opérateur `?.` est bien écrit avant les parenthèses.

3 Contrôle du flux

3.1 La condition : if

La condition `if` permet de d'exécuter une instruction (ou un bloc) si une expression est vraie.

Combiné à `else` et / ou `else if` il permet d'effectuer des tests en cascade.

	<pre>if (age >= 18) { console.log('adulte'); }</pre>
	<pre>if (age >= 18) { console.log('adulte'); } else { console.log('enfant'); }</pre>

Il est possible d'effectuer plusieurs évaluations en cascade avec `else if` :

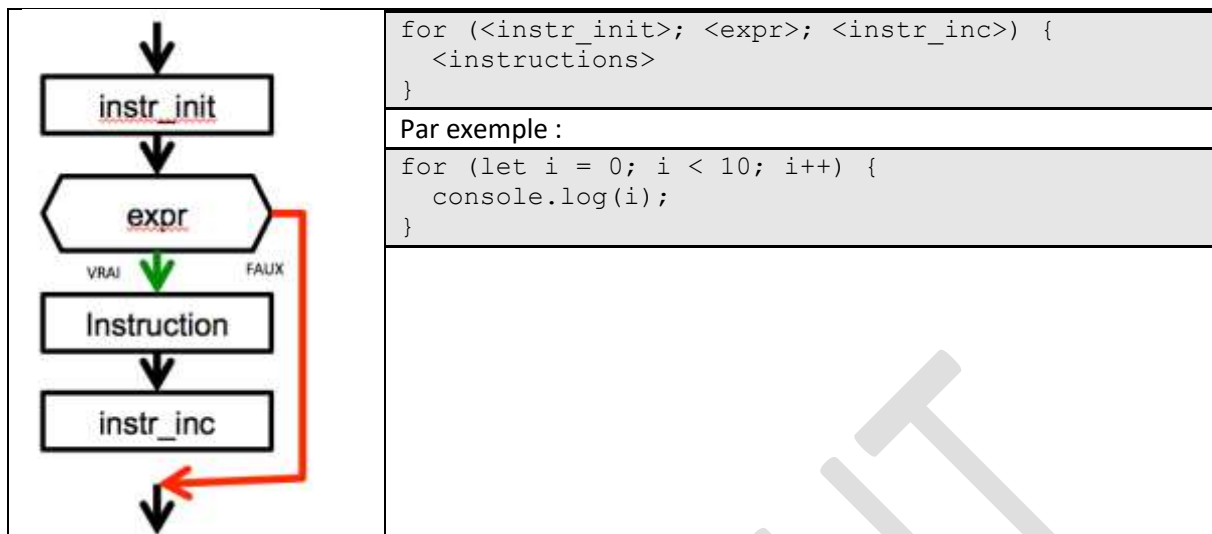
```
let angle;  
if ( angle == 90 ) {  
  angle = 'droit';  
} else if ( angle < 90 ) {  
  angle = 'aigu';  
} else {  
  angle = 'obtus';  
}
```

3.2 Le choix : switch

Le `switch` permet de diriger vers une instruction ou une autre en fonction de la valeur de l'expression évaluée. La clause `default` sera exécutée si l'expression évaluée n'est pas trouvée dans les clauses `case`. L'instruction `break` permet de sortir immédiatement d'un `switch` ou d'une boucle en cours.

```
let provenance, fruit = 'banane';  
switch (fruit) {  
  case 'banane':  
  case 'ananas': provenance = 'étranger';  
    break;  
  case 'pomme': provenance = 'normandie';  
    break;  
  default: provenance = 'inconnue';  
}
```

3.3 La boucle for



3.4 La boucle for...in

Elle permet de parcourir les propriétés (hors symboles) d'un objet itérable, c'est-à-dire un objet dont les propriétés sont énumérables (Array, Map, Set, String, TypedArray, etc...). Les propriétés sont récupérées successivement sous forme de string.

Attention ! L'ordre d'itération sur les propriétés peut varier

```
const personne = { nom: 'Smith', prénom : 'John' };  
for (let key in personne) {  
  console.log( 'Propriété trouvée : ' + key );  
}
```

```
Propriété trouvée : nom  
Propriété trouvée : prénom
```

Attention ! Appliquée sur un tableau, elle en fournit les index (clés), mais aussi toute autre propriété, un tableau étant un objet.

3.5 La boucle for...of

Introduite par ES6, elle permet aussi de parcourir certains objets itérables (Array, String, Map, WeakMap, Generator, NodeList, arguments).

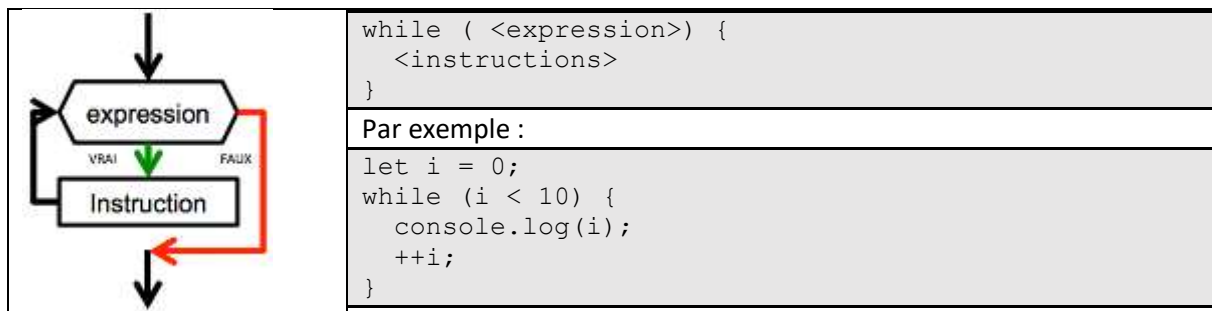
Par rapport à `for...in`, `for...of` fournit un ordre d'itération constant, et les index ne sont pas systématiquement convertis en string.

Le parcours est pour les tableaux sur chaque index (clé), pour les `string` sur chaque caractère, pour les `Maps` sur chaque clé, pour les `Sets` sur chaque élément, etc...

```
const liste = [3, 5, 2, 1]; // liste est un tableau de 4 éléments  
for (let val of liste) {  
  console.log( val );  
}
```

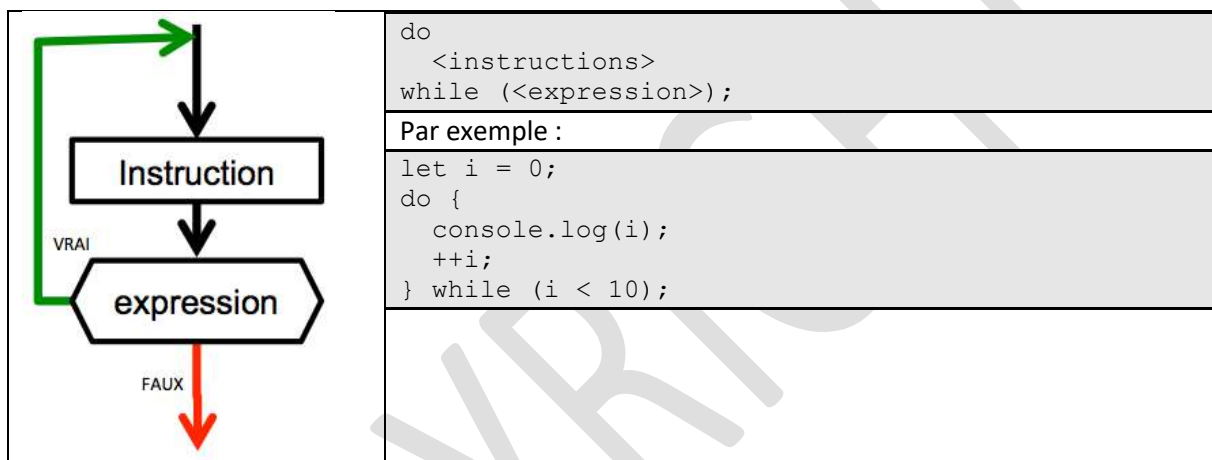
```
Affiche les valeurs 3, 5, 2, 1 en console
```

3.6 La boucle while



3.7 La boucle do .. While

Très similaire à la boucle `while` à part que l'instruction sera toujours exécutée au moins une fois.



3.8 L'instruction continue

De même que l'instruction `break` permet de sortir prématurément d'une boucle, l'instruction `continue` permet de directement aller à l'itération suivante de la boucle.

On peut toujours se passer de cette instruction mais elle peut permettre de simplifier l'écriture de structures imbriquées désagréables à lire parfois.

```
let s = 'Bonjour', ch = '';  
for (let i = 0 ; i < s.length; i++) {  
  if (i%2 == 0) continue;  
  ch += s.charAt(i);  
}  
console.log(ch);
```

oju

3.9 L'instruction return

Cette instruction n'a de sens que dans une fonction (qui seront abordées un peu plus loin). `return` provoque l'abandon de la fonction en cours et le retour à la fonction appelante, éventuellement en retournant une valeur définie. L'appel de la fonction sera évalué avec la valeur de l'expression retournée. Si aucune valeur n'est retournée, la fonction retournera `undefined`.

4 Tableaux

4.1 Principe

Un tableau est un objet `Array`. Il contient une liste de zéro, un ou plusieurs éléments indexés à partir de 0.

4.2 Écriture littérale

Un littéral tableau s'écrit entre crochets `[]` et `[]`. Chaque valeur est séparée par une virgule `,`.

```
[ ]           // expression tableau sans élément
[1, 3, 7]     // expression tableau de 3 éléments (1, 3, 7)
[1, , 7]      // expression tableau de 3 éléments (1, undefined, 7)
[1, 3, ]      // expression tableau de 2 éléments (1, 3)
```

Pour limiter les risques d'erreur, je vous conseille de déclarer explicitement les éléments valant `undefined` et d'éviter toute virgule inutile.

Un tableau est un objet. On peut donc l'affecter à une variable.

```
let a = [1, 3, 7];
```

4.3 Accès aux éléments

La propriété `length` fournit le nombre d'éléments :

```
console.log(['a', 'b', 'c'].length); // affiche 3
```

Il est facile d'accéder à un élément d'un tableau à partir de son index avec la notation `tableau[index]`.

```
[1, 3, 7][0] // expression valant 1
[1, , 7][1]  // expression valant undefined
a[1]         // 3 (avec a défini comme ci-dessus)
```

On peut aussi rechercher l'index du premier élément contenant une valeur :

```
[1, 3, 3].indexOf(3) // expression valant ici 1
[1, 2, 3].indexOf(4) // expression valant -1 (pas de correspondance)
```

4.3.1 Opérateur spread et tableaux

Depuis ES6, l'opérateur spread noté `...` permet de développer un objet itérable (un `Array` par exemple) :

```
[a, b, ...suite] = [1, 2, 3, 4, 5] // a = 1; b = 2 ; suite = [3, 4, 5]
const d = [2018, 12, 25];
const noel = new Date(...d);
```

Autre exemple, comme un objet `String` est itérable :

```
const ch = "Bonjour!"
let a = [...ch]; // a = ['b', 'o', 'n', 'j', 'o', 'u', 'r', '!']
```

Concaténer avec l'opérateur spread :

```
const a = [1, 2, 3];
```



```
const b = [...a, 4, 5, 6]; // b = [1, 2, 3, 4, 5, 6]
```

4.4 Parcours

Parcourir un tableau peut se faire de différentes manières. Les 2 lignes qui suivent afficheront les valeurs 1, 3 et 7.

```
let a = [1, 3, 7];
for (let i = 0; i < a.length; i++) { console.log(a[i]); }
for (let valeur of [1, 3, 7]) { console.log(valeur); }
```

Notons enfin `for ... in` qui permet de parcourir non seulement les éléments mais de façon générale les propriétés. Par exemple :

```
const a = {}
a.maPropriété = 'Bonjour';
for (let clé in a) { console.log(clé); }
```

La fonction `.forEach()` permet d'itérer sur chaque élément en leur appliquant une fonction que l'on fournit en argument. Revenez sur cet exemple, une fois lu le chapitre sur les fonctions :

```
let a = [1, 3, 7];
a.forEach(function(elem) { console.log(elem) });
```

4.5 Quelques méthodes utiles

4.5.1 Méthodes à effet de bord

Certaines méthodes modifient le contenu du tableau :

```
delete a[2]; // Supprime l'élément d'index 2
a.push(8); // Ajouter un élément valant 8 à la fin du tableau
a.pop(); // Supprimer le dernier élément et le retourne
a.unshift(9); // Ajouter un élément valant 9 au début du tableau
a.shift(); // Supprimer le premier élément et le retourne
```

La méthode `splice()` permet de supprimer et d'insérer des éléments dans le tableau :

```
a.splice(2, 3); // Supprime 3 élément à partir de l'index 2
a.splice(0, 2, 'd'); // Supprime 2 éléments et insère 'd' au début
```

4.5.2 Méthodes sans effet de bord

Certaines méthodes permettent de construire un nouveau tableau sans modifier l'original.

Les exemples qui suivent utilisent parfois une fonction en paramètre. Ce n'est pas grave si vous ne comprenez pas tout en première lecture. Relisez après la lecture du chapitre sur les fonctions, cette partie va devenir beaucoup plus claire.

4.5.2.1 Méthode `slice()`

La fonction `slice()` duplique le tableau :

```
let b = a.slice(); // Duplication du tableau a dans b
```

4.5.2.2 Méthode `reduce()`

La fonction `reduce()` permet de retourner une valeur unique à partir du tableau. La méthode de calcul doit être fournie en argument comme une fonction qui peut admettre 4 paramètres.

Par exemple pour calculer la somme des éléments du tableau :

```
const a = [1, 2, 3, 4];
let accumulateur = 0;
for (let i = 0; i < a.length ; i++) {
  accumulateur = accumulateur + a[i];
}
console.log(accumulateur);
```

En utilisant la méthode `reduce()`, ce code devient :

```
function f(accumulateur, valeurCourante) {
  return accumulateur + valeurCourante;
}
console.log( [1, 2, 3, 4].reduce(f) );
```

La fonction `f` est déclarée puis passée en paramètre de `reduce`. `f` admet 2 paramètres nommés `accumulateur` et `valeurCourante`. Pour chaque élément, `reduce` va appeler `f(<valeur de accumulateur>, <valeur de l'élément considéré>)`. La valeur retournée à la fin est la valeur finale de l'accumulateur.

Comme on ne fournit pas de valeur initiale à l'accumulateur, le premier appel sera `f(<premier élément>, <second élément>)`. Sinon le premier appel sera `f(<accu initial>, <premier élément>)`.

Les appels successifs seront donc :

```
// Nous n'avons pas fourni de valeur initiale à l'accumulateur
accumulateur = 1; // le premier élément
// On démarre donc au 2e élément qui vaut 2
accumulateur = f(1, 2); // accumulateur <- 1 + 2
accumulateur = f(accumulateur, 3); // accumulateur <- 3+3
accumulateur = f(accumulateur, 4); // accumulateur <- 6+4
```

Notez, que la notation fléchée montre ici tout son intérêt :

```
console.log( [1, 2, 3, 4].reduce((accu, valeur) => accu + valeur));
```

4.5.2.3 Méthode `map()`

La méthode `map()` retourne un tableau correspondant au tableau d'origine après application d'une fonction sur chaque élément :

```
[1, 2, 3, 4].map(n => n*2) // [2, 4, 6, 8 ]
```

4.5.2.4 Méthode `filter()`

La fonction `filter()` permet de créer un nouveau tableau contenant tout ou partie des éléments du tableau d'origine. La fonction en paramètre doit renvoyer `true` pour les valeurs à conserver :

```
[1, 2, 3, 4].filter(n => n%2 == 0 ) // [2, 4]
```

4.5.2.5 Déclaration et affectation par décomposition

Depuis ES6, on peut affecter les données venant d'un tableau dans différentes variables en une étape. Ceci s'appelle affectation par décomposition (*destructuring*) :

```
let a, b, suite;
[a, b, c] = [1, 2, 3]; // a = 1; b = 2; c = 3
```

```
[a, , c] = [1, 2, 3];           // a = 1; c = 3 La seconde valeur est ignorée  
[a,b,] = [1, 2, 3];           // Erreur de syntaxe  
let x, y;  
[x, y] = [1, 2];               // x = 1 ; y = 2;  
[x, y] = [y, x];               // échange des valeurs  
[x, y, [z]] = [1, 2, [3]];     // x = 1 ; y = 2 ; z = 3
```

On peut aussi déclarer les variables par la même occasion :

```
let [x, y] = [1, 2];           // x = 1; y = 2; après déclaration
```

5 Les fonctions

5.1 Principe

Une fonction contient une séquence d'instructions qui seront exécutées lorsque la fonction est appelée (on dit aussi lancée ou exécutée ou encore invquée).

Une fonction en JavaScript est un objet de première classe (on dit aussi fondamental). Ceci signifie qu'on peut manipuler une fonction comme tout objet sans restriction. Parmi les conséquences, une fonction peut :

- Avoir des propriétés, des méthodes
- Être le résultat d'une expression
- Être retournées par une fonction
- Être assignée à une variable
- Être envoyée comme argument (on parle de callback) à l'appel d'une autre fonction
- ...

5.2 Définition des fonctions

5.2.1 Définition de fonction par déclaration

JavaScript permet de définir une fonction de plusieurs manières.

On peut définir la fonction avec le déclarateur de fonction `function`. Le nom de la fonction sert à l'invoquer mais peut aussi servir comme expression valant l'objet fonction :

```
function calc(x, y) { // Définition de la fonction ❶
  return 2*x+y;      // retour de valeur
}
console.log(calc(2, 3)); // invocation, affiche 7 ❷
let f = calc;          // objet calc affecté à la variable f ❸
console.log(f(4, 5));  // invocation de calc à travers f, affiche 13 ❹
```

Dans ce qui précède, la fonction nommée `calc` est définie en ❶. Elle admet 2 paramètres nommés `x` et `y` et retourne une valeur calculée par l'expression `x + y`.

La ligne ❷ appelle la fonction en lui envoyant 2 arguments 2 et 3, puis affiche le résultat sur la console.

La ligne ❸ assigne la fonction `calc` (ici le nom est utilisé comme expression) à la variable `f`.

La ligne ❹ montre comment invoquer une fonction contenue dans une variable.

Toute fonction étant un objet `Function`, on pourrait déclarer une fonction avec le constructeur `Function()`, mais ce n'est pas recommandé.

5.2.2 Définition de fonction par expression

On peut aussi affecter une variable d'une expression qui vaut une fonction. On peut alors invoquer la fonction à partir de l'identificateur de la variable :

```
let f = function calc(x, y) { console.log(2*x+y); };
f(4, 5); // Appel de la fonction calc via la variable f
```

Le nom de la fonction dans l'expression est souvent inutile puisqu'on utilisera plutôt la variable pour y accéder. On peut donc créer une fonction sans nom (fonction anonyme) :

```
let db = function (x) { return 2*x; }; // affectation à db d'une fonction anonyme
```

ES6 propose en plus une syntaxe raccourcie avec l'expression de fonction fléchée. Le mot-clé `function` n'est plus indispensable :

```
let f = (x, y) => { return 2*x+y; };
```

NB : l'objet `this` n'existe pas dans une expression de fonction fléchée.

Une fonction dont le corps ne contient qu'un `return` peut s'écrire de façon encore plus simple :

```
let f = (x, y) => 2*x+y; // équivaut à => {return 2*x+y;}
```

Une fonction à paramètre unique peut s'écrire de façon simplifiée :

```
let db = a => { return 2*a }; // paramètre unique, plus de parenthèses
```

Et en combinant les simplifications :

```
let db = a => 2*a; // équivaut à : let db = function (a){return 2*x+y;};
```

5.2.3 Définition de fonctions immédiatement invoquées

JavaScript permet aussi de définir et d'immédiatement invoquer une fonction. On parle alors de IIFE pour Immediately Invoked Function Expression.

Plusieurs syntaxes le permettent, notez les parenthèses :

```
( function (x, y) {return 2*x+y;} )(1, 2);  
( function (x, y) {return 2*x+y;} (1, 2) );  
~function (x, y) {return 2*x+y;}(1, 2); // ou en préfixant avec +, -, !
```

Cette façon de faire était très utile avant ES6, quand les déclarations de variables se faisait avec `var`, qui, rappelons-le, déclare en fait une propriété de l'objet global. Il n'y avait donc pas de notion de portée, et les variables étaient globales. Afin de profiter de la notion de portée de variable dans une fonction, on utilisait souvent des fonctions pour créer un environnement fermé, fonctions qui étaient immédiatement invoquées. JS proposait donc un sucre syntaxique pour simplifier cette écriture.

Avec l'avènement de `let` et `const`, les IIFE sont moins utiles maintenant.

5.2.4 Définition de fonction imbriquée

Une fonction étant un objet, elle peut contenir des méthodes, donc des fonctions imbriquées dans la fonction :

```
function f(x) {  
  function g(y) {  
    return 2*y + x;  
  }  
  return g(y) + 1;  
}
```

5.2.5 Les fermetures

On peut rendre une fonction imbriquée accessible à l'extérieur de la fonction. On appelle alors la fonction imbriquée fermeture (ou closure) en ce sens qu'elle conserve un « environnement » de données propres : tant que la fonction existe, son environnement aussi. Ainsi, même si la fonction « mère » est finie, son environnement persiste avec les closures.

```
function creerMultiplicateur(x) {
  return function (y) { return x * y; }; // On retourne une fonction
}
const mult10 = creerMultiplicateur(10); // mult10 contient une fonction
let mult100 = creerMultiplicateur(100); // mult100 contient une fonction
console.log(mult10(3) + mult100(4));
```

Dans l'exemple ci-dessus, `mult10` est une variable qui contient une fonction anonyme (retournée par `creerMultiplicateur()`) qui prend un paramètre, le multiplie par 10, et retourne le résultat. Donc `mult10(3)` est un appel de la fonction $y \Rightarrow y \times 10$ avec l'argument 3. Le résultat est donc 30. Le même raisonnement tient pour `mult100(4)` qui retourne 400, et le résultat final est 430.

Ces deux fonctions sont des closures puisque la valeur de `x` qui a permis de créer la fonction `mult10` (`x = 10` dans ce cas) existe encore lors de l'appel de la fonction anonyme stockée dans la variable `mult10`, ce qui permet d'effectuer l'opération $x \times y$, où `x` est connu dans la closure (et vaut 10) et `y` est le paramètre de la fonction anonyme. Si la fonction anonyme ne conservait pas son environnement, elle ne connaîtrait plus la valeur de `x` utilisée lors de sa construction.

5.3 Valeur de retour

L'instruction `return <expression>;` stoppe l'exécution de la fonction et retourne la valeur de l'expression. La valeur de retour peut être de n'importe quel type.

Une fonction qui se termine sans `return`, ou avec un `return` sans expression retournera `undefined`.

Avec l'affectation par décomposition (destructuring), on peut simuler un retour multi-valué, i.e. le retour de plusieurs valeurs :

```
function f() { return [1, 2]; }
let a, b;
[a, b] = f(); // vaut [a, b] = [1, 2] donc a ← 1 et b ← 2
let x = f(); // x ← [1, 2] et est donc un tableau
console.log(x[0]); // 1
[,b] = f(); // On ignore la première valeur donc b ← 2
```

5.4 Paramètres et arguments

5.4.1 Différence entre paramètre et argument

Un peu de vocabulaire, car les notions de paramètre et d'argument étant proches, elles sont parfois confondues par abus de langage. Le paramètre correspond à une variable locale à la fonction alors que l'argument est une expression évaluée lors de l'appel de la fonction et dont la valeur sera affectée au paramètre correspondant.

```
function f(x) { return 3*x; } // Fonction définie avec un paramètre x
f(4+1);                     // Fonction appelée avec l'argument 4+1
                             // x ← 5 au début de l'appel.
```

En général, une fonction est invoquée avec un nombre d'arguments correspondant au nombre de paramètres.

```
function f(x) { return 2*x; } // 1 paramètre x
console.log(f(3)); // Nb d'arguments = nb de paramètres, affiche 6
```

5.4.2 Fonctions à nombre d'arguments variables

Une fonction peut être invoquée avec un nombre d'arguments différent du nombre de paramètres définis.

Les arguments en trop ne seront simplement pas affectés à un paramètre nommé :

```
console.log(f(3, 8)); // second argument non utilisé, affiche 6
```

À l'inverse, la valeur `undefined` sera assignée à tout paramètre nommé dont l'argument manque :

```
console.log(f()); // Pas d'argument, retourne NaN (= 2*undefined)
```

Les arguments en trop sont néanmoins accessibles dans l'objet `arguments`. Ce mécanisme permet de définir des fonctions à nombre de paramètres variable.

Dans ce qui suit, on construit un tableau `args` à partir de l'objet `arguments` pour en simplifier l'utilisation :

```
function cumul() {
  let ret = 0;
  let args = Array.prototype.slice.call(arguments); // Récup sous forme d'array
  for (let i in args) {
    ret += args[i];
  }
  return ret;
}
console.log(cumul(1, 2, 3) + cumul(4, 5)); // 6 + 9 => 15
15
```

ES6 fournit une syntaxe simplifiée pour convertir l'objet `arguments` en un tableau :

```
let args = Array.from(arguments);
```

L'opérateur spread (ES6) donne une autre façon d'accéder aux arguments qui seraient en surnombre.

```
La fonction qui suit retourne la valeur du premier argument plus le nombre d'arguments reçus ensuite.
function f(x, ...autresArgs) {
  return x + autresArgs.length;
}
console.log(f(1) + " / " + f(4, 'a', 1));
1 / 6
```

NB : L'objet `arguments` n'existe pas dans les fonctions fléchées.

5.4.3 Valeur par défaut

Avant ES6, pour donner une valeur par défaut à un attribut, on était obligé de ruser :

```
function f(x) {
  x = (x === undefined) ? 0 : x;
  return 2 * x;
}
```

Depuis ES6, il est possible de fixer la valeur par défaut d'un paramètre. C'est-à-dire la valeur que prendra le paramètre quand il reçoit `undefined` lors de l'appel :

```
function f(x = 0) { return 2 * x ;} // x ← 0 par défaut (si pas d'argument)
console.log(f(1));                // x ← 1. Affiche 2
console.log(f());                 // x ← 0 (défaut). Affiche 0
console.log(f(undefined));        // x ← 0 (défaut). Affiche 0
```

5.4.4 Passage par variable

Le passage de paramètres se fait par valeur, ce qui signifie que le paramètre reçoit une copie de la valeur de l'argument. Ceci implique qu'une variable utilisée comme argument n'est pas modifiée par la fonction, puisque seule sa valeur est transmise à la fonction et stockée dans une variable locale à la fonction (le paramètre).

```
function f(x) {
  x = 2*x;
  return x;
}
let x = 10;
let y = f(x);
// Ici x vaut 10, y vaut 20
```

5.4.5 Passage d'un objet

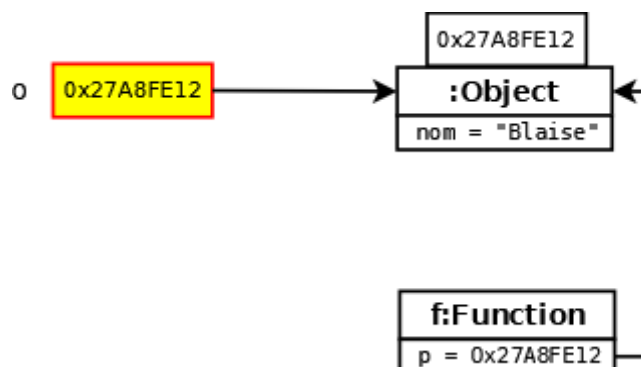
L'invocation d'une fonction avec un objet comme argument se fait par référence. C'est-à-dire que le paramètre sera initialisé avec la référence de l'objet. En fait, on passe la valeur de l'argument, mais cette valeur est bien la référence à l'objet. Donc le paramètre reçoit la référence lors de l'appel de la fonction.

Donc le contenu de l'objet peut être modifié dans le corps de la fonction.

C'est le même principe que l'utilisation d'un `const` pour déclarer un objet : on ne peut plus modifier la référence, mais on peut modifier l'objet.

Dans l'exemple suivant `f()` modifie l'objet référencé par la variable `o`, mais ne modifie jamais le contenu de la variable `o` (c'est-à-dire la référence de l'objet) :

```
function f(p) {
  p.nom = "Blaise";
}
const o = {nom: "Bob"}; // o peut être une constante, f ne modifiera pas sa valeur
console.log(o.nom); // "Bob"
f(o); // f modifie la propriété 'nom' de o, pas la valeur de o
o.prénom = "Vincent"; // Là encore on ne modifie pas la variable o !
console.log(o.prénom + ' ' + o.nom); // "Vincent Blaise"
// o = { prénom: "Vincent"}; // ERREUR car là on modifie o !
```



5.5 Contexte d'exécution

5.5.1 Principe

Chaque appel de fonction génère un contexte d'exécution qui consiste en :

- Un environnement lexical : les paramètres de la fonction, les variables déclarées par la fonction et les autres fonctions déclarées dans cette fonction
- Un lien vers le contexte d'exécution de la fonction dans laquelle elle est déclarée² et ainsi de suite³
- Un objet `this` qui représente l'objet qui a invoqué la fonction

Il existe un contexte d'exécution en dehors de toute fonction, appelé contexte d'exécution global.

5.5.2 Remontée automatique des déclarations (hoisting)

Le contexte d'exécution est analysé et généré AVANT l'exécution des instructions de la fonction. Lorsque le code de la fonction commence à s'exécuter, JavaScript -via le contexte- a connaissance de toute variable, de toute fonction qui y est déclarée, même plus tard dans le code :

```
function f() {  
  a = 1; // Affectation avant la déclaration. Correct en JavaScript !!!!!  
  var a;  
}
```

On parle de hoisting (remontée ou *hissage* en anglais) car cela ressemble à une « remontée » des déclarations.

Ceci explique aussi pourquoi un appel de fonction peut se situer AVANT sa définition :

```
f(1); // Appel avant la définition, correct en JavaScript !  
function f(x) {  
  return 2*x;  
}
```

Cela ne concerne pas les affectations qui sont, elles, effectuées à l'exécution :

```
function f() {  
  console.log(a); // Correct mais a vaut undefined ET PAS 1  
  var a = 1; // Maintenant a vaut 1  
}
```

Dans le cas d'une variable déclarée avec `let` ou `const`, une utilisation anticipée déclenchera une `ReferenceError`.

```
console.log(a); // Erreur ReferenceError  
let a = 3;
```

Je vous conseille de toujours déclarer vos variables avant de les utiliser.

² Ce n'est pas nécessairement celle dans laquelle elle est invoquée.

³ On parle de chaîne de contexte (scope chain)

5.5.3 Impact sur la portée des variables

La portée (scope) d'une variable c'est l'emplacement dans le code où on peut y accéder (lecture / écriture).

5.5.3.1 Portée d'une variable déclarée avec var

Une variable déclarée avec `var` en dehors d'une fonction a une portée globale et est donc accessible et modifiable dans tout le programme.

```
var a = 'Bonjour'; // a est de portée globale, i.e. accessible partout
f(); // Affiche "Bonjour" et modifie la valeur de a
console.log(a); // Affiche "Au revoir", la nouvelle valeur de a
function f() {
  console.log(a); // a est accessible car de portée globale
  a = 'Au revoir'; // Modification de a
}
```

Une variable déclarée avec `var` dans une fonction a pour portée cette fonction. Si cette variable a même nom qu'une variable globale elle masque temporairement cette dernière.

```
var a = 'Bonjour'; // a est de portée globale
f(); // Affiche "Au revoir Maurice"
console.log(a); // Affiche "Bonjour". La variable globale a n'a pas changé
console.log(b); // ReferenceError car b n'existe pas en dehors de f()
function f() {
  var a = 'Au revoir'; // masquage temporaire de a dans la fonction
  var b = 'Maurice'; // b est déclaré pour la fonction
  console.log(a, b); // affiche "Au revoir Maurice"
}
```

Une fonction `g` imbriquée dans une fonction `f` contenant la déclaration d'une variable aura accès à cette variable via la scope chain :

```
var a = 1; // Contexte d'exécution global={a, h(), f()}
// a variable globale
h(2); // 6
function h(n) { // h est définie globalement
  // Son contexte d'exécution (CE)={n, d}
  // est chaîné au CE global={a, h(), f()}
  var d = a + n; // Donc h peut accéder à n et a (par la chaîne)
  f(d); // et peut aussi appeler f() (par la chaîne)
}
function f(p) { // f est définie globalement
  // Son CE={p, b, g()} est chaîné
  // au CE global {a, h(), f()}
  function g() { // g est définie dans f
    // Son CE={b} est chaîné au CE de f {p, b, g()}
    var b = 2; // Masquage de b (celui dans f())
    console.log(a + b + p); // Accès à a (CE global), b (CE de g), et p (CE de f)
  }
  var b = a + p; // f Accède à p (CE local) et à a (CE global)
  console.log(b); // 4
  // ++d; // ReferenceError car d n'est pas dans la scope chain
  g();
}
```

5.5.3.2 Portée d'une variable déclarée avec `let` ou `const`

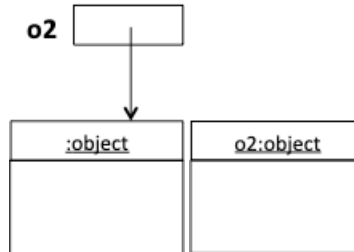
Une variable déclarée avec `let` ou `const` a une portée de bloc. Elle est accessible entre les accolades du bloc et dans tout sous-bloc.

```
let a = 'Bonjour';
f(); // Affiche "Au revoir Mauricette"
console.log(a); // Affiche "Bonjour"
console.log(b); // ReferenceError car b n'existe pas ici
function f() {
  let a = 'Au revoir'; // Masquage temporaire du a global
  let b = 'Maurice'; // locale à la fonction
  if (true) {
    let a = 'Bye'; // Masquage temporaire du a local à la fonction
    let i = 1; // i est locale au bloc if
    b = b + 'tte'; // b est accessible dans le sous-bloc if
  }
  console.log(a); // c'est le a local à la fonction
  for (let a = 1; a < 10; ) { // Masquage temporaire du a de la fonction
    a++; // c'est le a local au bloc for
  }
  i++; // ReferenceError car i n'existe pas ici
  console.log(b); // 'Mauricette'
}
```

6 Les objets

6.1 Notation

Maintenant que nous avons bien intégré qu'une variable n'est pas l'objet qu'elle contient, on peut simplifier la représentation en mémoire. Dans la suite, les 2 notations suivantes sont équivalentes :



6.2 Méthodes

6.2.1 Rappel

Nous avons vu qu'un objet est un ensemble de propriétés nommées et qu'on pouvait créer un objet littéral facilement avec la notation `{ pté1 : valeur1, pté2 : valeur2, ... }`.

```
const o = { nom : 'Dupont' }; // Affectation d'un objet à la variable o
Une propriété affectée d'une fonction s'appelle une méthode :
const o = {
  saluer : function() { return 'Bonjour'; }
};
```

Appeler la méthode d'un objet se dit aussi invoquer la méthode sur l'objet.

```
console.log(o.saluer());
```

6.2.2 Syntaxe alternative

ES5 fournit une notation plus courte pour définir une méthode. Sans utiliser cette notation on écrit :

```
const o = {
  saluer : function() { return 'Bonjour'; }
};
```

En utilisant la notation simplifiée, on écrit :

```
const o = {
  saluer() { return 'Bonjour'; }
};
```

6.2.3 Accesseurs (getters)

JavaScript permet de créer une sorte de propriété dynamique avec le mot-clé `get` associé au nom de cette propriété, et au code qui sera exécuté lors d'un accès à cette propriété. C'est aussi une façon de reporter le moment où la valeur d'une propriété doit être calculée au moment où elle est appelée.

```
const o = {
  get maProp() { <calculs> ; return <valeur>; }
};
```

La pseudo-propriété ainsi créée est accessible comme toute autre :

```
console.log(o.maProp);
```

```
console.log(o['maProp']);
```

6.2.4 Mutateurs (setters)

De façon similaire on peut créer une pseudo-propriété dont l'accès en modification lancera l'exécution d'un code donné. On utilise alors le mot-clé `set` :

```
const o = {  
  languages : [],  
  set lang(val) { this.languages.push(val); }  
};  
o.lang = 'FR';  
o.lang = 'EN';  
console.log( o.languages ); // Array ['FR', 'EN']
```

6.3 Instanciation par constructeur

6.3.1 Fonction constructeur

Nous allons maintenant décrire des façons de créer des objets sans utiliser la notation littérale.

On peut créer un objet avec l'opérateur `new` appliqué à une fonction appelée alors constructeur.

Nous l'avons fait auparavant en créant un objet avec le constructeur natif `Object()`.

```
const o = new Object(); // Affectation à o d'une instance d'Object.
```

6.3.2 Constructeur Object()

L'intérêt du constructeur `Object()` est de pouvoir créer des objets à partir de valeurs de type primitif :

```
const o1 = new Object();           // objet vide  
const o2 = new Object(null);       // objet vide  
const o3 = new Object(undefined);  // objet vide  
const o4 = new Object(123);        // objet de type Number  
const o5 = new Object("abc");      // objet de type String  
const o6 = new Object(true);       // objet de type Boolean
```

6.3.3 Constructeur wrapper

Nous avons aussi évoqué que JavaScript fournit des constructeurs pour chaque type primitif (attention à la première lettre en majuscule) : `Number()`, `String()`, `Boolean()`, `BigInt()`, etc.

Ces constructeurs permettent de créer directement des objets du type voulu à partir d'expressions :

```
let o = new Boolean(true);          // Objet Boolean à partir d'un boolean  
let o = new String("machin");      // Objet String à partir d'une string
```

6.3.4 Création d'un objet via le constructeur : new

Un constructeur étant une fonction, on peut tout à fait définir notre propre constructeur.

Le nom d'un constructeur prend une majuscule par convention pour distinguer constructeurs et simples fonctions.

```
function Voiture(){}              // Notre constructeur Voiture()  
let o = new Voiture();             // Crée un nouvel objet et l'affecte à o
```

Le code du constructeur est exécuté à chaque instanciation avec `new`.

Lors d'une instanciation avec `new`, voici le déroulement dans JS :

- Création d'un objet
- Exécution du code du constructeur
- Retour de l'objet créé.

```
function Voiture() {
  console.log('Le code du constructeur a bien été exécuté !');
}
console.log('Un objet a été créé : ' + (typeof new Voiture() ===
'object'));
```

```
Le code du constructeur a bien été exécuté !
Un objet a été créé : true
```

Attention à ne pas oublier `new` pour instancier un objet, sinon la fonction s'exécute comme attendu :

```
console.log("La fonction appelée sans new retourne : " + Voiture());
```

```
La fonction appelée sans new retourne : undefined
```

6.3.5 Accès à l'objet courant dans le constructeur : `this`

Le code du constructeur a souvent besoin d'accéder à l'objet créé avant qu'il ne soit retourné. Le mot clé `this` permet d'y accéder en lecture comme en écriture. Ceci permet d'ajouter et d'initialiser des propriétés à l'instanciation pour adapter notre constructeur à notre besoin :

```
function Voiture(m) {
  this.marque = m;
  this.compteur = 0;
  this.roule = function(km) {this.compteur += km;};
  this.afficheInfo = () => `Marque: ${this.marque}, Compteur:
${this.compteur} km`;
}
const o = new Voiture('Opel'); // assigne à o un objet de type Voiture
console.log(o.compteur);
o.roule(1000); // Appel de la méthode rouler
console.log(o.afficheInfo ());
```

```
0
Marque: Opel, Compteur: 1000 km
```

Dans l'exemple précédent l'objet `o` est instancié avec le constructeur `Voiture()` qui lui associe quatre propriétés (`marque`, `compteur`, `rouler` et `afficheInfo`), dont les deux dernières sont des méthodes (elles contiennent une fonction).

6.4 Prototypes

6.4.1 Propriété interne `[[prototype]]` d'un objet

Chaque objet a systématiquement une propriété notée `[[prototype]]` (`__proto__`) qui référence un objet ou `null`.

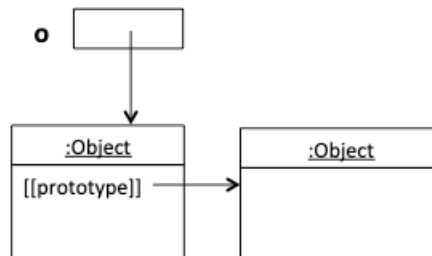
C'est une propriété dite *interne* car elle n'est pas accessible comme les autres propriétés avec l'opérateur point `.`. Il est nécessaire de passer par les fonctions `Object.setPrototypeOf()` et `Object.getPrototypeOf()` pour y accéder.

Pour simplifier l'écriture je noterai `o.[[prototype]]` la valeur de cette propriété étant bien entendu que cette syntaxe n'a pas de sens à être écrite dans un code puisque la propriété est interne.

Exemple :

```
let o = { };
console.log("Le prototype de o est un objet : " +
  (typeof Object.getPrototypeOf(o) === "object"));
```

Le prototype de o est un objet : true



Nous allons en voir l'intérêt (héritage de propriétés) plus loin. Pour l'instant comprenons le mécanisme.

6.4.2 Propriété prototype des objets function

Rappelons en préambule qu'un constructeur est une fonction et qu'une fonction est un objet `function`. Donc un constructeur peut avoir ses propriétés comme tout objet.

Par exemple un objet `function` a une propriété appelée `name` qui vaut le nom de la fonction sous forme de `string`.

Les objets `function` ont systématiquement une propriété nommée `prototype`. Cette propriété est importante car elle désigne l'objet à utiliser comme valeur de `[[prototype]]` pour tout objet créé avec la fonction.

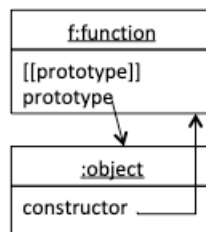
L'objet référencé par cette propriété a lui-même systématiquement une propriété `constructor` référençant la fonction.

ATTENTION ! Ne pas confondre la propriété interne `[[prototype]]` qui existe pour tout objet et la propriété `prototype` qui n'existe que pour les objets `function`.

```
function f () { }
console.log(f.prototype.constructor.name);
```

f

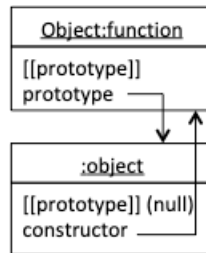
Nous allons voir d'autres exemples après avoir vu quelques objets importants.



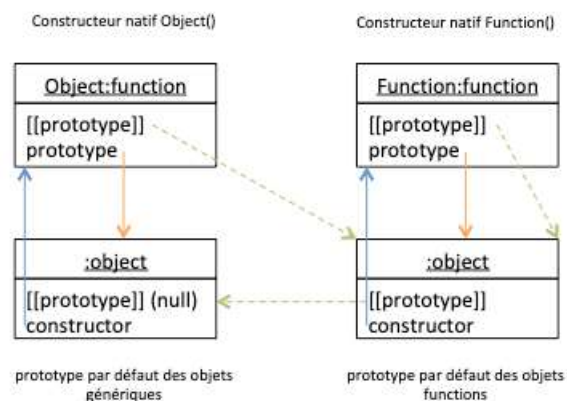
6.4.2.1 Les objets `Object.prototype` et `Function.prototype`

Un objet particulier va revenir souvent dans ce cours, c'est l'objet référencé par la propriété `prototype` du constructeur natif `Object()` (qui est bien une fonction et possède donc cette propriété). Remarquons tout de suite que la valeur de `[[prototype]]` pour cet objet est `null`. Cet objet agit comme un prototype de base pour tout objet et fournit des méthodes comme `toString()`, ou `valueOf()`.

```
console.log(Object.getPrototypeOf(Object.prototype));  
null
```



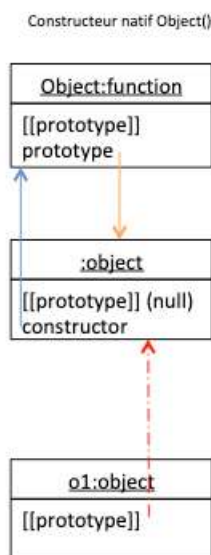
Un autre objet important est l'objet référencé par la propriété `prototype` du constructeur de fonction natif `Function()`. Voici schématisé ci-dessous ces objets et leurs relations :



6.4.2.2 Exemples

Dans le cas de la création d'un objet par littéral, le constructeur implicite est `Object()`. Donc `o.[[prototype]]` va référencer le même objet que `Object.prototype`.

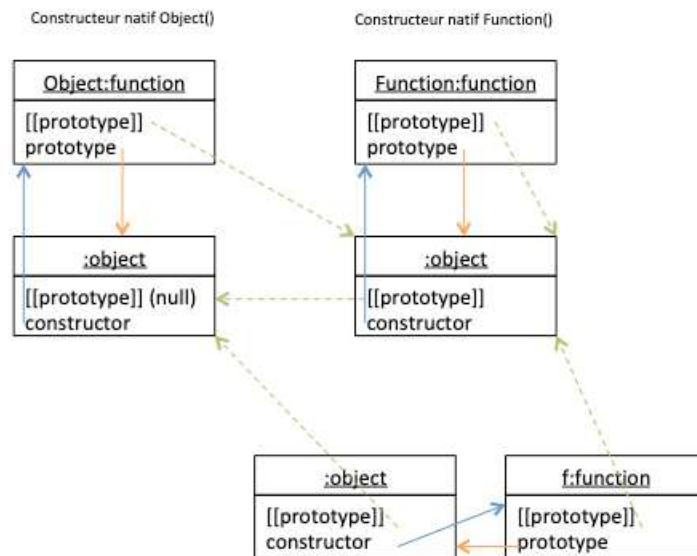
```
const o1 = { };
```



Dans le cas de la création d'une fonction `f`, le constructeur implicite est `Function()` donc `f.[[prototype]]` va référencer le même objet que `Function.prototype`. Par ailleurs, en tant que fonction, `f` aura aussi une propriété `prototype` initialisée à un nouvel objet qui aura lui-même une propriété `constructor` référençant `f` en retour. Ce nouvel objet (étant créé avec le constructeur

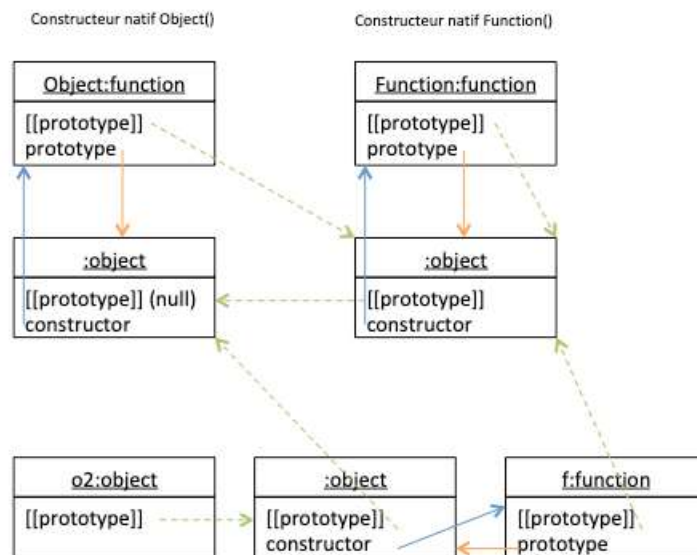
implicite `Object()`), aura sa propriété `[[prototype]]` référençant l'objet référencé par `Object.prototype`.

```
function f() {}
```



Si on crée un objet `o2` avec `f` comme constructeur, là encore la règle va s'appliquer et `o2.[[prototype]]` va référencer le même objet que `f.prototype`.

```
function f() {}  
const o2 = new f();  
console.log(Object.getPrototypeOf(o2).constructor.name);  
f
```



6.4.3 Intérêt de la propriété `[[prototype]]`

L'objectif de cette propriété `[[prototype]]` est de pouvoir déléguer la recherche d'une propriété inexistante dans un objet à son prototype et le cas échéant au prototype de son prototype, etc... jusqu'à celui qui aura sa propriété `[[prototype]]` à `null`. On parle de recherche dans la chaîne des prototypes.

On dit qu'un objet hérite des propriétés de son prototype.

Quand une propriété est utilisée, JavaScript cherche cette propriété en cascade dans la chaîne de prototypes et prend la première valeur trouvée, ou `undefined` sinon.

6.4.4 Exemples

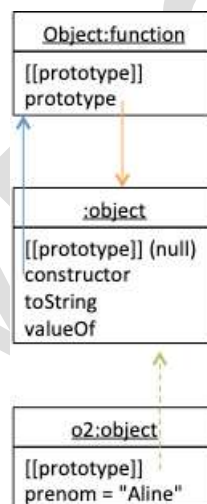
6.4.4.1 Exemple 1

Nous l'utilisons depuis longtemps sans nous en rendre compte comme le montre l'exemple suivant :

```
const o2 = { prenom : 'Aline' };
console.log(o2.prenom);
console.log(o2.nom);
console.log(o2.toString());
console.log(typeof o2.valueOf() === "object");
```

```
Aline
undefined
[Object object]
true
```

La situation en mémoire peut être schématisée comme suit :



La propriété `o2.prenom` est immédiatement trouvée dans l'objet `o`.

La propriété `nom` n'est pas trouvée dans `o2`, ni dans `o2.[[prototype]]`. Et `o2.[[prototype]].[[prototype]]` vaut `null` ce qui marque la fin de la chaîne donc de la recherche. La valeur affichée est donc `undefined`.

La méthode `toString()` n'existe pas dans `o2`, mais existe dans son prototype. La méthode `toString()` du prototype sera donc exécutée et retournera une `string` représentant l'objet.

Idem pour `valueOf()` qui retournera l'objet référencé par `o2`.

6.4.4.2 Exemple 2

Nous allons voir comment modifier la chaîne de prototype avec `setPrototypeOf()` pour faire hériter un objet des propriétés d'un autre :

```
const o1 = { type : 'eleve' }; // ❶
const o2 = {}; // ❷
Object.setPrototypeOf(o2, o1); // ❸
```

```

console.log(o2.type === 'eleve');           // ④
console.log(o2.toString());                 // ⑤
Object.getPrototypeOf(o2) === o1;           // true

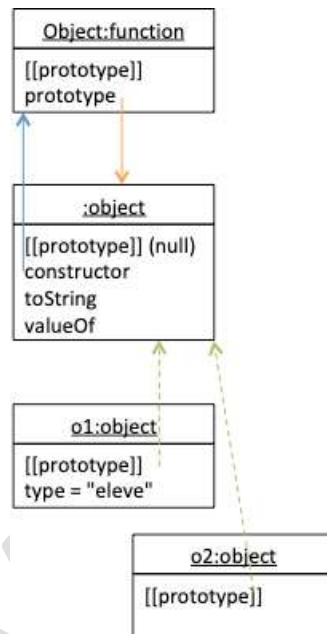
```

En ligne ①, nous créons un objet (avec une propriété `type`) que nous assignons à la variable `o1`.

En ligne ②, nous créons un objet sans propriété que nous assignons à la variable `o2`.

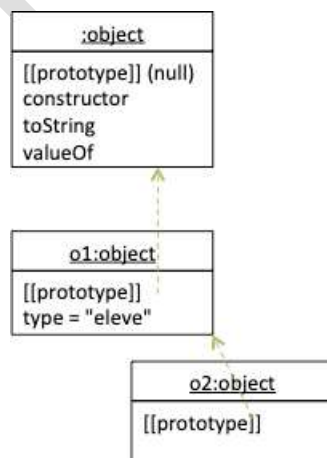
Ces objets ont été créés par littéraux (donc implicitement avec le constructeur `Object()`). Donc

`o1.[[prototype]]` et `o2.[[prototype]]` référencent le même objet que `Object.prototype`.



En ligne ③, nous assignons `o1` comme prototype de `o2`.

En ligne ④, nous accédons à la propriété `o2.type`. JavaScript ne trouve pas `type` dans les propriétés propres de `o2` alors il parcourt la chaîne de prototypes. Il trouve `type` dans le premier objet suivant (`o1`). L'expression `o2.type === 'eleve'` est donc évaluée à `true` et affichée sur la console.



En ligne ⑤, la fonction `toString()` est invoquée à partir de l'objet `o2`. JavaScript cherche et trouve cette fonction dans la chaîne des prototypes. La fonction `toString()` peut alors être exécutée et son résultat `'[object Object]'` est affiché.

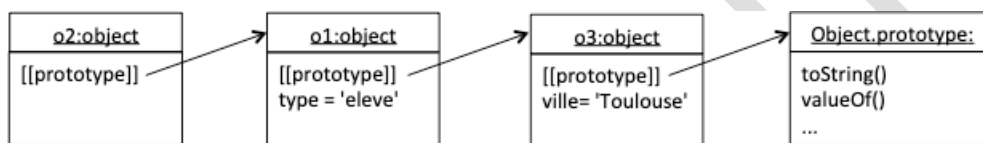
Notons enfin que la propriété `constructor` est aussi héritée de la même façon. Sa valeur est l'objet `function Object`.

```
o2.constructor === o2.constructor && o1.constructor === Object // true
```

6.4.4.3 Exemple 3

Voici un exemple de chaîne de prototype un peu plus longue :

```
const o1 = {type : 'eleve'};
const o2 = {};
const o3 = {ville : 'Toulouse'};
Object.setPrototypeOf(o2, o1);
Object.setPrototypeOf(o1, o3);
o2.type == 'eleve' // true
o1.ville == 'Toulouse' // true
o2.ville == 'Toulouse' // true
```



Les objets `o2`, `o1`, `o3` ont accès à la propriété `ville`. Les objets `o1` et `o2` ont accès à la propriété `type`.

6.4.5 Propriétés propres et propriétés héritées

La fonction `hasOwnProperty(<propriété>)` retourne `true` si la propriété est propre à l'objet et `false` si elle est héritée via la chaîne de prototypes.

Rappelons que les propriétés d'un objet sont accessibles avec une boucle `for...in`.

Par exemple, le code ci-dessous exécuté sur la situation de l'exemple précédent donnera :

```
for (let prop in o1) {
  if (o1.hasOwnProperty(prop)) {
    console.log('Propriété propre : ' + o1[prop]);
  } else {
    console.log('Propriété héritée : ' + o1[prop]);
  }
}
```

```
Propriété propre : eleve
Propriété héritée : Toulouse
```

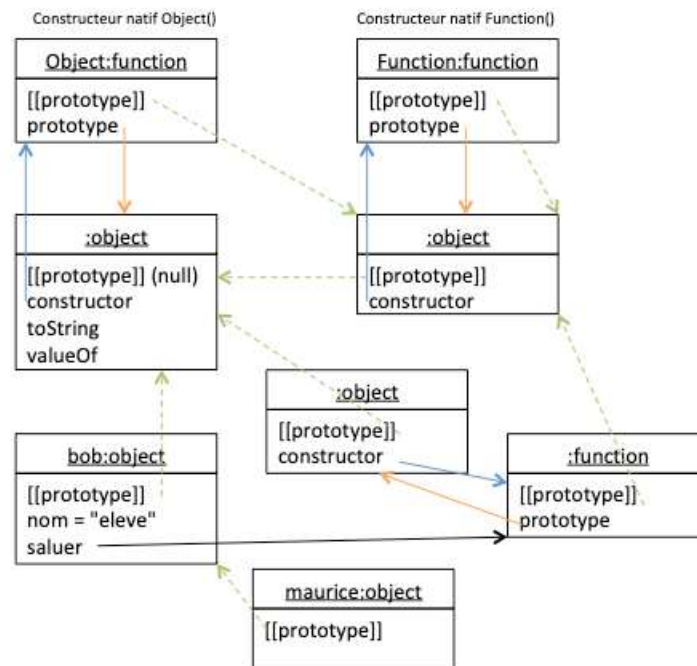
Par ailleurs, les propriétés propres sont accessibles dans un tableau retourné par `Object.keys(o1)`.

6.5 Instanciation par prototype

On peut créer un objet à partir d'un autre qui sera son prototype avec la méthode `Object.create()` :

```
const bob = {
  nom : 'Bob',
  saluer : function() { return "Je m'appelle " + this.nom ; }
}
const maurice = Object.create(bob);
maurice.nom = 'Maurice';
console.log( maurice.saluer() );
```

Le code ci-dessus aboutira à la situation en mémoire suivante :



Par la suite, on n'aura plus besoin de représenter tous les objets. Juste les principaux concernés par les explications.

6.6 Plus loin avec les prototypes : héritage

6.6.1 Méthode de prototype

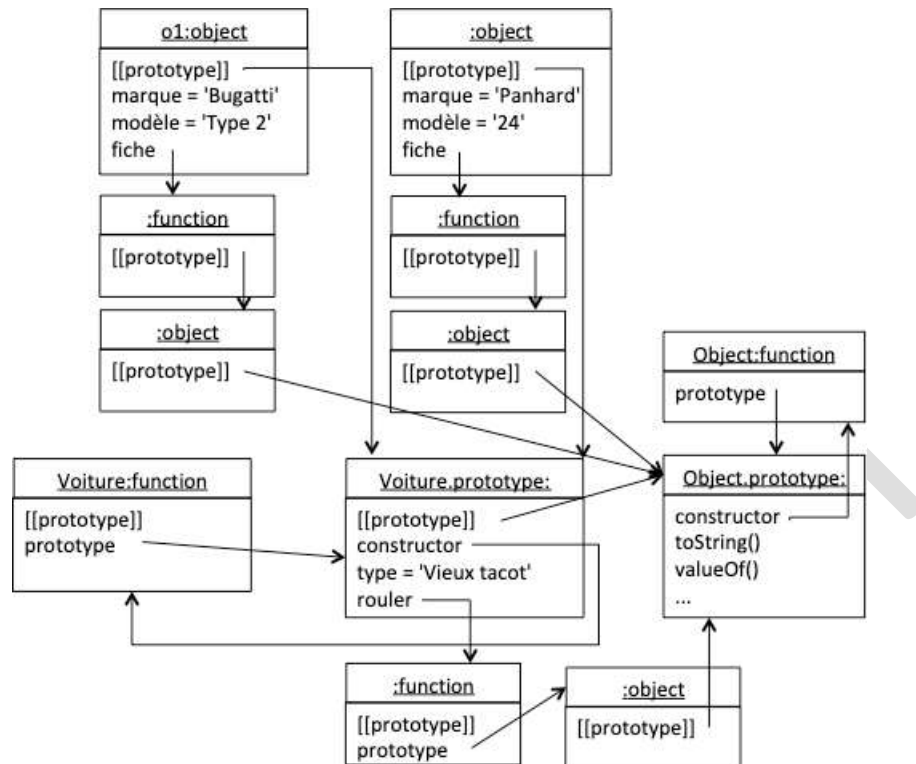
On peut ajouter / modifier des propriétés directement dans le prototype.

L'avantage est que tous les objets qui héritent du prototype vont bénéficier des changements :

```
function Voiture(marque, modèle) {
  this.marque = marque;
  this.modèle = modèle;
  this.type = 'Vieux tacot';
  this.compteur = 0;
  this.fiche = () => 'Marque: ' + this.marque
    + '; Modèle: ' + this.modèle
    + '; Type: ' + this.type
    + '; Compteur: ' + this.compteur;
};
Voiture.prototype.rouler = km => { this.compteur += km; };
Voiture.prototype.type = 'Vieux tacot';
const o1 = new Voiture('Bugatti', 'Type 2');
const o2 = new Voiture('Panhard', '24');
o1.rouler(500);
o2.rouler(1000);
console.log(o1.fiche());
console.log(o2.fiche());
```

Marque: Bugatti; Modèle: Type 2; Type: Vieux tacot; Compteur: 500
 Marque: Panhard; Modèle: 24; Type: Vieux tacot; Compteur: 1000

Pour tester votre bonne compréhension de ce qui précède, je vous conseille vivement de prendre le temps d'essayer de construire par vous-même le diagramme de la situation en mémoire après l'exécution du code ci-dessus avant de regarder ci-dessous.



Ça pique un peu, mais rassurons-nous, nous n'aurons pas besoin d'avoir tout à l'esprit tout le temps. Il suffit de penser que nous manipulons deux objets de type `Voiture`. Ces objets ont des propriétés propres et héritent de propriétés via leur prototype.

Notez la différence importante entre les méthodes `fiche` et `rouler`. Il y a un objet `Function` différent pour chaque méthode `fiche` (même si le code de la fonction est le même). Alors que l'objet `Function` pour `rouler` est le même. Ceci explique pourquoi il est en général préférable de définir les méthodes dans le prototype.

6.6.2 Utilisation de `this`

Le mot-clé `this` peut sembler complexe quand on est habitué à un autre langage. Son sens change en fonction du contexte. Les principaux cas de figure sont résumés ci-dessous, mais vous en saurez plus en allant voir :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/L_op%C3%A9rateur_this.

Comme nous l'avons vu auparavant, utilisé dans un constructeur, il fait référence à l'objet instancié :

```
function Voiture(){
  this.type = 'Tacot';
};
```

Similairement, dans une méthode invoquée sur un objet, `this` fait référence à cet objet :

```
function Voiture(){
  this.fiche = () => { return 'je suis de marque ' + this.marque; }
}
```

```
v = new Voiture('Mercedes');
v.marque = 'Mercedes';
console.log (v.fiche());
```

Dans le contexte global, en dehors de toute fonction, il fait référence à l'*objet global* :

```
this === window // true dans un navigateur
```

Dans une fonction simple, en mode non strict, il fait référence à l'*objet global* :

```
function f() {
  return this;
}
f() === window // true dans un navigateur
```

6.6.3 Méthodes call et apply

La méthode `call()` permet d'exécuter une fonction en lui passant en premier paramètre la valeur que prendra `this` dans cette fonction. Dans le chapitre qui suit, nous en voyons une utilisation pratique.

```
Voiture.call(<valeur de this>, 'Bugatti', 'Type 2');
```

Notez qu'il existe aussi une méthode sensiblement identique appelée `apply()`. La différence tient en ce que `apply()` n'accepte que deux arguments, la valeur que doit prendre `this` et un tableau de valeurs.

6.6.4 Un exemple d'héritage

Supposons que dans notre application, nous voulions modéliser des personnes par un *nom* et un *prénom*. Les personnes ont aussi la capacité à se *présenter* en affichant leur nom et prénom.

Nous souhaitons aussi manipuler des *élèves* qui auront toutes les caractéristiques d'une *personne* mais avec une propriété en plus, leur *niveau d'étude* (6e, 1re, terminale, etc.).

On pourrait redéfinir des propriétés `nom` et `prénom` propres à un objet `Eleve`.

Comme les évolutions futures de `Personne` devraient être applicables à `Eleve` (par exemple l'ajout d'une propriété comme la *date de naissance*) nous choisissons plutôt que `Eleve` hérite de `Personne`.

Nous allons donc mettre en prototype d'`Eleve` un objet qui sera chaîné sur `Personne.prototype`.

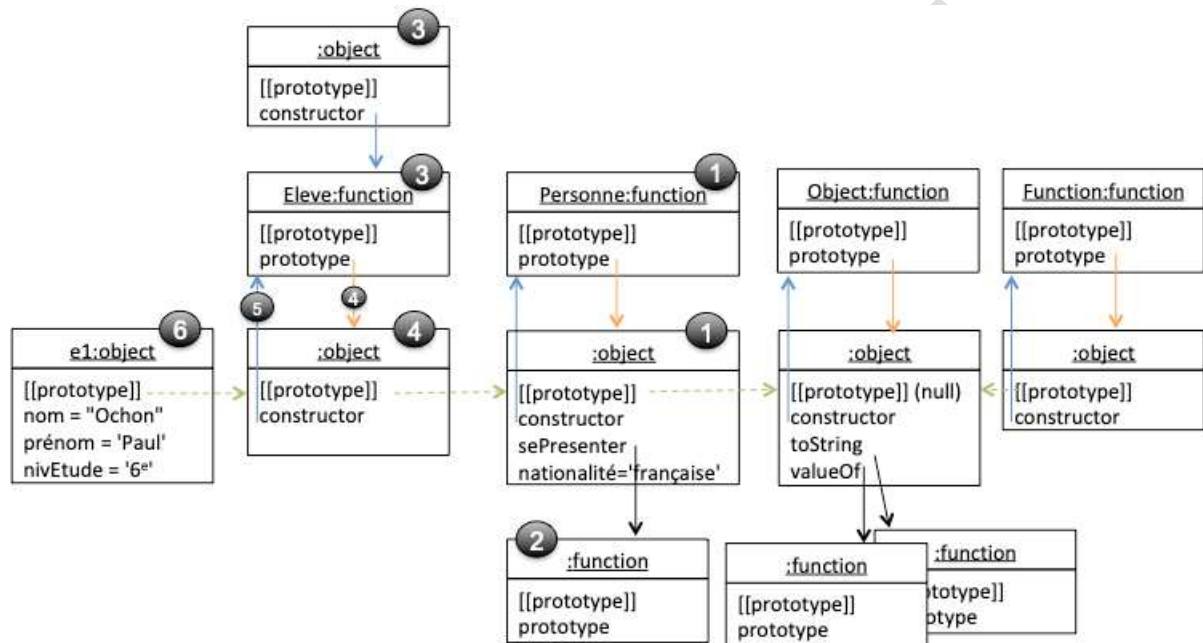
```
// Constructeur personne
function Personne(nom, prénom) { // ❶
  this.nom = nom;
  this.prénom = prénom;
}
Personne.prototype.sePresenter = function() {
  return this.prénom + ' ' + this.nom
}; // ❷
// Constructeur Eleve
function Eleve(nom, prenom, nivEtude) { // ❸
  Personne.call(this, nom, prenom);
  this.nivEtude = nivEtude;
}
Eleve.prototype = Object.create(Personne.prototype); // ❹
Eleve.prototype.constructor = Eleve; // ❺
```

```
const e1 = new Eleve('Ochon', 'Paul', '6e'); // 6
console.log(e1.sePresenter()); // 7
Personne.prototype.nationalité = 'française'; // 8
console.log(e1.nationalité); // 9
```

```
Paul Ochon
française
```

En ❶, nous définissons un constructeur `Personne`. Sa propriété `prototype` référence un objet `Object.prototype`.

Voici le schéma de ce qui se passe :



Sur le schéma ci-dessus toutes les références n'ont pas été représentées par des flèches pour ne pas alourdir. Notons simplement que les propriétés `[[prototype]]` des objets référencent le même objet que `Object.prototype` et que les propriétés `[[prototype]]` des fonctions référencent le même objet que `Function.prototype`.

En ❶, nous déclarons la fonction `Personne`. Ceci a pour effet de créer son prototype.

En ❷ nous ajoutons à son prototype la méthode `sePrésenter`.

En ❸, nous déclarons la fonction `Eleve`. Ceci a pour effet de créer son prototype.

En ❹, nous modifions `Eleve.prototype` pour qu'il référence maintenant un nouvel objet créé pour l'occasion. Cet objet a pour prototype `Personne.prototype`.

En ❺, nous nous assurons que ce nouvel objet a bien une propriété `constructor` référençant `Eleve`.

En ❻, nous assignons à la variable `o1` un nouvel objet créé avec le constructeur `Eleve`.

En ❼, nous appelons sur `o1` la méthode `sePresenter()` qui est trouvée via la chaîne des prototypes.

En ❽, nous ajoutons la propriété `nationalité` au prototype de `Personne`.

En ❾, nous accédons à cette propriété sur l'objet `o1` via la chaîne des prototypes.

6.7 Les classes

6.7.1 Pourquoi ?

Depuis ES6, on peut définir des « classes » avec une syntaxe proche de Java.

Ces classes d'ES6 ne sont pas un nouveau concept. JavaScript reste un langage basé sur les prototypes qui ne possède pas le concept de classe tel qu'on l'entend en Java. Il s'agit simplement d'une nouvelle écriture pour simplifier la création des objets et la gestion des héritages (on parle de *sucre syntaxique*). Au passage, les développeurs venant d'autres langages retrouvent certaines marques.

L'idée est d'englober dans une classe le constructeur d'objet et les données et comportements qui lui sont associés.

6.7.2 Définition d'une classe

En JavaScript, une classe n'est qu'une fonction un peu spéciale. On peut donc comme toute fonction la définir par déclaration ou par expression.

On déclare une classe avec le mot clé `class`, un nom, et un corps (un bloc de code entre accolades) :

```
class Voiture {} // Définition de la classe Voiture
```

Par convention un nom de classe prend une majuscule.

Une expression de classe s'écrit similairement à une expression de fonction. Et pareillement on peut créer une expression de classe anonyme :

```
const Tacot = class TacotNom {} // Expression de classe nommée  
const Tacot = class {} // Expression de classe anonyme
```

Nommer une classe permet d'y faire référence dans son corps :

```
let Tacot = class TacotNom {  
  TacotNom.name  
}
```

6.7.3 Instanciation

L'opérateur `new` crée un nouvel objet à partir du nom de la classe (on dit *instancier* la classe et l'objet est appelé une *instance* de la classe) :

```
class Voiture {} // Déclaration de la classe Voiture  
let instanceVoiture = new Voiture(); // instanciation
```

Similairement dans le cas d'une expression de classe :

```
const Tacot = class {} // Affectation d'une expression de classe anonyme  
const instanceTacot = new Tacot(); // Instanciation
```

6.7.4 Remontée (hoisting)

Contrairement aux fonctions, il n'y a PAS de remontée pour la définition d'une classe. Une classe doit donc être écrite dans le code avant son utilisation :

```
let v = new Voiture; // ReferenceError: la classe n'est pas encore définie  
class Voiture {}
```

6.7.5 Corps d'une classe

Le corps d'une classe comprend ses propriétés, le constructeur et les méthodes.

Le corps d'une classe est exécuté en mode strict.

6.7.5.1 Constructeur

La fonction constructeur appelée à l'instanciation s'appelle systématiquement `constructor()`.

Il ne doit y avoir qu'une méthode avec le nom `constructor` par classe (sous peine de `Syntax Error`) :

```
class Voiture {  
  constructor() { }  
}
```

6.7.5.2 Méthodes

Les fonctions invocables sur une instance sont appelées les méthodes de la classe. Avec cette écriture, `rouler()` est une méthode associée au prototype et un objet `function` ne sera donc pas créé pour chaque instance.

```
class Voiture {  
  rouler() { }  
}
```

On retrouve alors une syntaxe proche de Java. Voici le code équivalent à celui de l'exemple précédent (à ceci près qu'ici `fiche()` et `rouler()` sont des méthodes de prototype) :

```
class Voiture {  
  constructor(marque, modèle) {  
    this.marque = marque;  
    this.modèle = modèle;  
    this.compteur = 0;  
  }  
  fiche() {  
    return 'Marque: ' + this.marque  
      + '; Modèle: ' + this.modèle  
      + '; Type: ' + this.type  
      + '; Compteur: ' + this.compteur;  
  }  
  rouler(km) {  
    this.compteur += km;  
  }  
}  
const o1 = new Voiture('Bugatti', 'Type 2');  
const o2 = new Voiture('Panhard', '24');  
o1.rouler(500);  
o2.rouler(1000);  
console.log(o1.fiche());  
console.log(o2.fiche());  
console.log(instanceOf(o1, Voiture));
```

Rappel : pas de hoisting pour les classes, il faut déclarer avant d'instancier.

6.7.5.3 Méthodes statiques

Le mot-clé `static` permet de définir une méthode statique, c'est-à-dire une méthode qui n'a pas besoin d'un objet pour être invoquée. On utilise la syntaxe `<nom de la classe>.méthode()` pour l'invoquer.

...

```

static inviter() { // méthode de classe
    return 'Bienvenue dans le monde de voitures';
}
...
console.log (Voiture.inviter()); // Appel à partir du nom de la classe

```

6.7.6 Héritage, sous-classe

Le mot-clé `extends` permet à une classe d'hériter d'une autre :

```
class classeFille extends classeMere { ... }
```

En programmation objet, cette situation est associée à plusieurs vocabulaires. On dira que la classe qui hérite :

- Spécialise⁴ (ou est une spécialisation de) celle dont elle hérite
- Est une classe fille⁵ de la classe dont elle hérite (appelée alors sa classe mère)
- Est une sous-classe de celle dont elle hérite (appelée alors la super-classe)

Dans le constructeur de la classe fille, on peut appeler le constructeur de la classe parente avec la fonction `super()`.

On peut aussi appeler une méthode parente avec l'objet `super`.

Dans l'exemple suivant on définit une classe `Eleve` qui hérite de la classe `Personne`. Les objets de la classe `Eleve` auront les propriétés de la classe `Personne` et en plus une propriété `nivEtude` :

```

class Personne {
    constructor(nom, prénom) {
        this.nom = nom;
        this.prénom = prénom;
    }
    sePrésente() {
        return this.prénom + ' ' + this.nom;
    }
}
class Eleve extends Personne {
    constructor(nom, prénom, classe) {
        super(nom, prénom); // Appel du constructeur de la classe mère
        this.nivEtude = classe;
    }
    sePrésente() {
        return super.sePrésente() + ' en ' + this.classe;
    }
}
const e1 = new Eleve('Térier', 'Alain', '5e');
console.log(e1.sePrésente());

```

Alain Térier en 5e

⁴ C'est le terme probablement le plus adapté en conception objet.

⁵ Cette appellation est dangereuse en particulier dans les langages permettant l'héritage multiple. Une classe se retrouverait alors fille de plusieurs mères ! Je ne recommande pas ce terme mais il est bon de le connaître car employé.

Comparer cet exemple avec ce qu'il avait fallu faire en termes de gestion des prototypes dans l'exemple similaire un peu avant montre l'intérêt d'utiliser les classes pour simplifier les écritures et la compréhension du code.

COPYRIGHT

7 Programmation avancée

7.1 Le débogage

7.1.1 Les types de bug

Un bug (ou en français boque) est un comportement anormal du code. On peut trouver deux grands types de bug : les bugs syntaxiques, et les bugs logiques (ou algorithmiques).

Les bugs syntaxiques viennent d'une erreur de frappe ou de syntaxe lorsqu'on tape le code :

```
varr maVar = "toto;
```

Ici on trouve deux bugs : mauvaise orthographe du mot-clé `var` et les guillemets ne sont pas fermés. Ces bugs sont faciles à trouver car l'interpréteur vous les indique (nous verrons comment).

Les bugs logiques viennent d'une erreur de votre part dans le fonctionnement même de votre programme. Mais l'interpréteur n'est pas dans votre tête, il ne peut pas savoir ce que vous voulez faire, il ne trouve donc aucune erreur, mais votre code ne fait pas ce qu'il devrait :

```
let cpt = 0;
while (cpt < 10) {
  console.log("coucou");
} // On n'en sort jamais !
```

Typiquement une erreur de logique : on entre dans la boucle, mais on ne modifie jamais la variable de fin de boucle. Donc la boucle est infinie et on n'en sort jamais ! Il suffit ici d'ajouter une incrémentation du compteur dans la boucle et tout rentre dans l'ordre :

```
let cpt = 0;
while (cpt < 10) {
  console.log("coucou");
  cpt++;
} // C'est mieux !
```

7.1.2 Le débogage

Le débogage consiste à supprimer les bogues de votre code. Pour les bugs syntaxiques, c'est l'interpréteur qui vous les signalera, merci à lui.

Pour les bugs logiques, en revanche, il faudra tracer le déroulement de votre code pas à pas en analysant à chaque ligne ce qui se produit. Les kits de développement vous y aideront. Cependant, avant d'en arriver là, il est fréquent d'utiliser des commandes comme `console.log()` pour afficher les valeurs qu'on soupçonne d'être fausses, ce qui permet déjà de cibler la partie du code qui pêche.

7.1.3 Les outils de développement

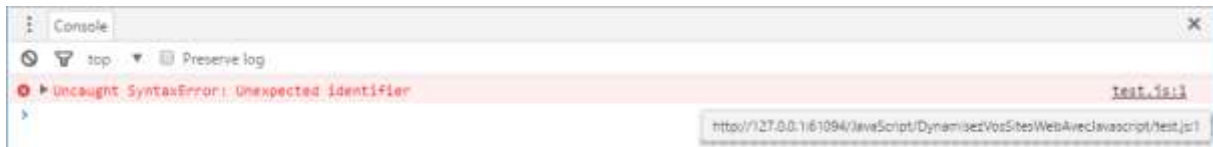
De nos jours, tout navigateur qui se respecte possède des outils de développement. Ces outils nous permettent de visualiser les valeurs des variables, leurs modifications, les requêtes HTTP et bien d'autres choses encore.

Pour afficher les outils de développement, appuyez sur la touche `F12` (en tout cas sur Firefox et Chrome sous Windows). Sinon vous les trouverez :

Sur Chrome dans le menu -> plus d'outils -> Outils de développement.

Sur Firefox dans le menu -> Développement -> Outils de développement.

Vous obtenez alors la console en bas de votre fenêtre. Si nous avons un code syntaxiquement faux, la console va nous l'afficher en rouge, ainsi qu'un lien cliquable nous amenant à l'endroit où l'interpréteur trouve l'erreur :



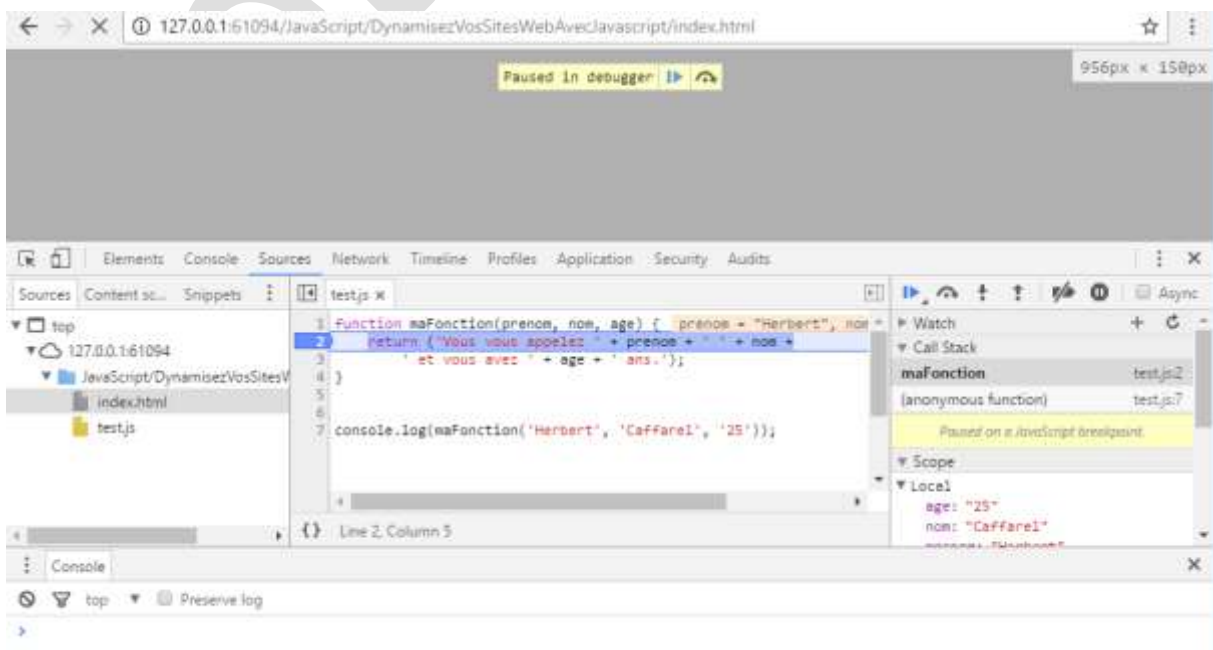
Il ne reste qu'à cliquer et voir ce qu'il se passe :



L'affichage change et propose maintenant une vue plus complète, incluant le code source au milieu, l'arborescence du serveur à gauche et la gestion du flux à droite.

Ici nous voyons bien sur quelle ligne on a le problème. Effectivement, le mot-clé `functin` n'existe pas, il est mal orthographié.

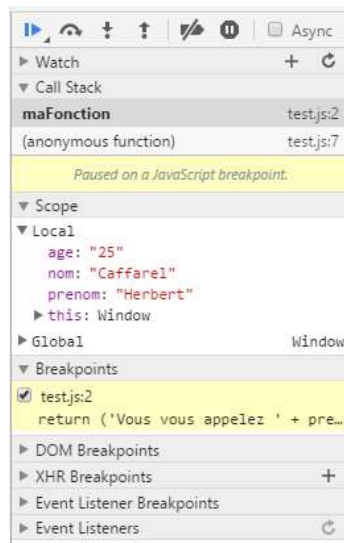
Pour les bugs un peu plus subtils, vous pouvez utiliser la gestion du flux en positionnant des points d'arrêt dans votre code directement en cliquant sur le numéro de ligne. En relançant votre code (donc en rechargeant la page), l'exécution va s'arrêter à cet endroit :



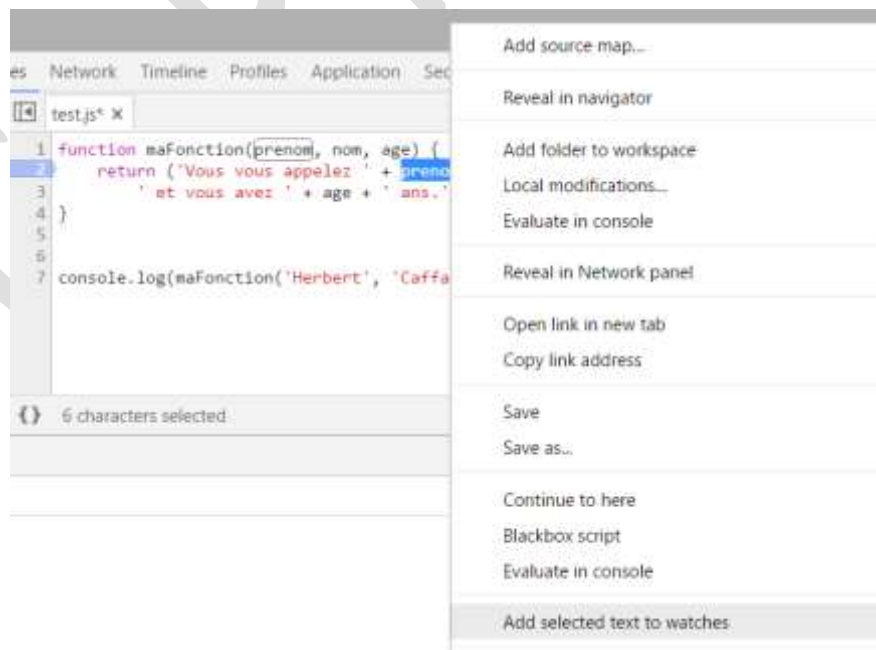
Vous pourrez ensuite avancer pas-à-pas avec les commandes de gestion de flux. Les icônes représentent, dans l'ordre :

- Reprendre le flux jusqu'au prochain arrêt.
- Exécuter la ligne suivante et s'arrêter. Si c'est l'appel d'une fonction par exemple, cette fonction sera exécutée et le curseur s'arrêtera à la ligne suivante.
- Entrer dans la prochaine commande et s'arrêter avant de la continuer. Si c'est l'appel d'une fonction, par exemple, le curseur se retrouvera au début de ladite fonction.
- Sortir de la fonction courante : le script s'exécute jusqu'à la sortie de la fonction et s'arrête.

On trouve également sous la gestion de flux tout un tas d'informations utiles :

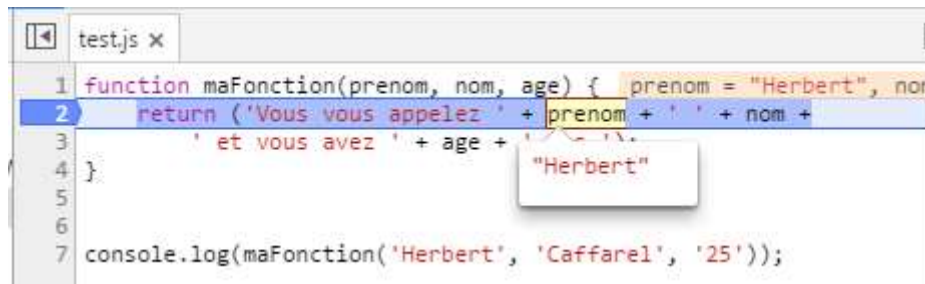


- Les informations des variables surveillées (ce qu'on appelle communément les espions). On peut surveiller une variable par simple clic droit sur elle, puis `Add selected text to watches`.



- L'empilement des appels de fonction depuis le script principal (la pile d'exécution).

- Les valeurs des variables locales. Mais on peut également voir les valeurs des variables locales simplement en passant la souris dessus, la valeur s'affiche en pop-up :



Avec ces outils, vous êtes armés pour analyser le comportement de votre code pas-à-pas.

Nota : on appelle souvent les outils de développement la console par abus de langage. En effet, les outils intègrent la console, mais sont loin de se limiter à elle. C'est pourtant souvent ainsi qu'on les nomme.

7.2 API, Plugins et Frameworks

Une API (Application Programming Interface) est un ensemble d'outils qui permet de faire communiquer entre eux plusieurs programmes ou langages.

Un plugin est une bibliothèque de fonctions mises ensembles dans un même paquet, permettant d'ajouter un ensemble de fonctionnalités à un logiciel ou un langage. On peut l'imaginer comme une boîte à outils.

Un Framework est une bibliothèque qui s'appuie sur un langage mais qui en change la façon de l'utiliser. Par exemple jQuery : c'est bien une bibliothèque JavaScript, mais c'est aussi un Framework car il introduit une nouvelle façon de programmer.

Il existe de nombreux plugins et frameworks pour JavaScript. Certains sont bien écrits, d'autres beaucoup plus douteux. Je ne peux que vous encourager à la plus grande vigilance quand vous intégrez un plugin à votre code. Faites beaucoup de tests et surtout vérifiez que le plugin répond effectivement à vos attentes. Il est parfois plus rapide de créer ses propres fonctions collant parfaitement à sa demande, plutôt que d'utiliser celles créées par un autre qui sont « presque » ce dont on a besoin.

Trouver des plugins n'est pas un problème, des sites comme <https://www.javascripting.com> par exemple les recensent. Trouver LE plugin qu'il vous faut peut devenir problématique. Vous pouvez y passer des heures. Donc j'insiste, il est parfois plus simple (ou en tout cas plus rapide) d'écrire sa propre boîte à outils.

8 Document Object Model

8.1 Introduction

8.1.1 Qu'est-ce que le DOM ?

Maintenant que nous savons utiliser le langage de façon basique, nous allons pouvoir nous intéresser à ce qui en a fait un incontournable du web, à savoir sa facilité d'interaction avec les pages HTML.

Pour interagir avec une page web, JavaScript s'appuie sur un objet particulier qui modélise la structure de la page web, et qui contient un ensemble de méthodes pour agir sur ses propriétés. Cet objet s'appelle *Document Object Model (DOM)*. C'est donc une API permettant de faire communiquer JavaScript avec HTML (et d'une manière plus générale avec XML).

8.1.2 Historique

Du temps des dinosaures, lorsque JavaScript a été intégré dans les navigateurs, chacun utilisait sa propre structuration du DOM, imposant au codeur des prouesses pour que son code fonctionne partout. Le W3C a mis de l'ordre dans tout ça en 1998 en imposant un standard, qu'on appelle *DOM-Level 1 core* ou encore *DOM 1*, spécifiant exactement la structuration d'un document HTML et plus généralement XML. Cette structure est celle d'un arbre hiérarchique : l'élément `<html>` contient deux enfants (les éléments `<head>` et `<body>`) qui contiennent à leur tour des enfants.

Une nouvelle spécification, appelée subtilement *DOM-2*, a ensuite été publiée, introduisant la merveilleuse fonction `getElementById()` que nous ne tarderons pas à voir.

Le *DOM-3* est apparu en 2004, apportant le support de XPath, mais surtout la gestion des événements claviers.

Le *DOM-4* est encore en cours de création, même si ses spécifications sont déjà bien avancées et souvent déjà supportées par les navigateurs les plus performants. La dernière spécification date en effet de novembre 2015...

Remarquez au passage que, si en HTML on parle de *paires de balises*, en JavaScript on parle d'*élément HTML*, simplement parce que chaque paire de balises est modélisée comme un objet à part entière dans le DOM.

Pour satisfaire votre insatiable curiosité, vous pouvez trouver les documents afférents aux différentes versions du DOM sur cette page : <https://www.w3.org/DOM/DOMTR>

8.1.3 L'objet window

Il est nécessaire, avant d'attaquer le DOM proprement dit, de connaître l'existence de l'objet `window`. Cet objet modélise la fenêtre du navigateur, c'est ce qu'on appelle un *objet global*. C'est donc un « sur-objet de la page elle-même. Et c'est à partir de lui que le JavaScript est exécuté.

Ainsi, la majorité des fonctions que nous avons pu utiliser, comme par exemple `alert()`, sont en réalité des méthodes de l'objet `window`. On n'a cependant pas besoin d'indiquer l'objet `window` qui est dit implicite. Ainsi, les deux lignes suivantes sont identiques :

```
window.alert("Coucou");  
alert("Coucou");
```

Attention cependant, toutes les fonctions ne sont pas nécessairement des méthodes de `window`. Il existe aussi des fonctions globales comme par exemple `isNaN()`, ou `parseFloat()`, ou encore `isFinite()` ou `escape()`. Mais de telles fonctions sont rares.

Je vous propose d'ailleurs de regarder l'objet `window` dans les outils de développement en ajoutant un espion dessus :



On y trouve toutes les méthodes existantes, dont `alert()`.

Tout ceci est finalement peu important. Ce qui l'est beaucoup plus, ce qu'il faut savoir, c'est que toute variable déclarée de façon globale est en fait attachée à l'objet `window`. Ainsi dans le code suivant on masque une variable globale en déclarant une variable locale du même nom, mais on peut toujours accéder à la variable globale comme propriété de l'objet `window` :

```
var text = 'Variable globale !';

(function() { // On utilise une IIFE pour isoler le code
  var text = 'Variable locale !';
  console.log(text); // Affiche 'Variable locale !'
  console.log(window.text); // Affiche 'Variable globale !'
})();
```

Enfin, et c'est fondamental, toute variable non déclarée est automatiquement attachée à l'objet `window`, *quel que soit l'endroit* où elle est utilisée. D'où le code suivant :

```
(function() { // IIFE
  text = 'Variable accessible !'; // Variable non déclarée
})();
alert(text); // Affiche 'Variable accessible !'
```

De toute façon, c'est une mauvaise idée de ne pas déclarer ses variables avant de les utiliser, donc vous ne le faites pas, n'est-ce pas ?

8.1.4 L'objet document

Si vous avez un peu parcouru l'objet `window` dans la console, vous avez pu noter l'existence de la propriété `document`. C'est un sous-objet de `window` qui représente la structure interne de la balise `<html>`. C'est donc grâce à cet objet que nous allons pouvoir accéder aux éléments constitutifs de la page web, et donc les modifier.

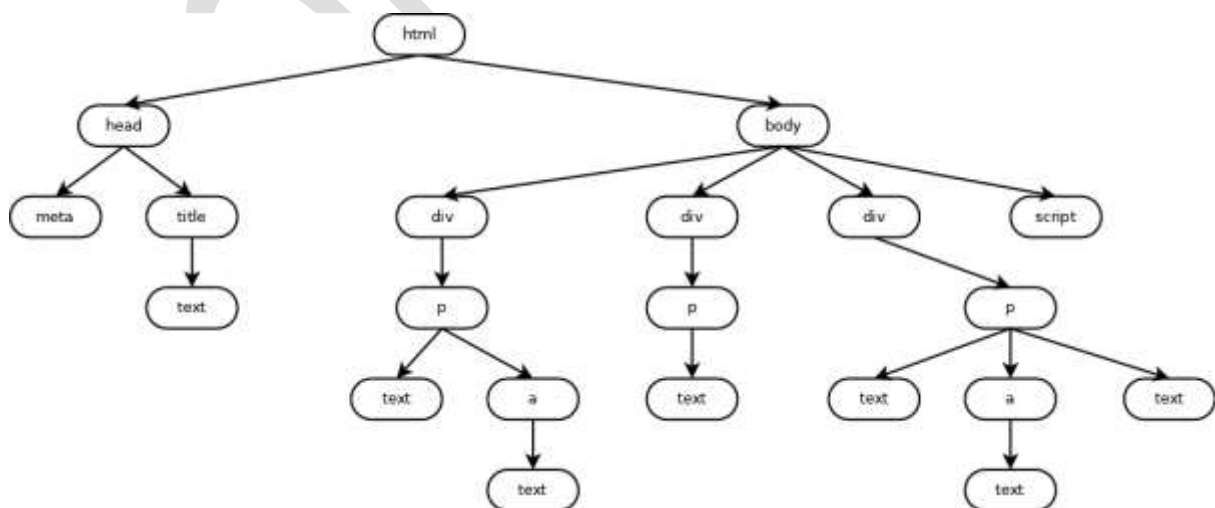
8.2 Accéder à la structure du DOM

Dans le DOM, la page web est vue comme un arbre hiérarchique d'éléments HTML.

Soit la page :

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>DOM sélection d'éléments</title>
</head>
<body id="body">
  <div id="div1">
    <p id="p1" class="rouge">Un paragraphe dans un div et <a id="a1"
href="#">un lien dans le paragraphe</a></p>
  </div>
  <div id="div2">
    <p id="p2">Un autre paragraphe</p>
  </div>
  <div id="div3">
    <p id="p3">Un troisième paragraphe et <a id="a2" href="#"
class="rouge">un second lien</a> et encore du texte</p>
  </div>
  <script></script>
</body>
</html>
```

Alors le DOM a cette forme :



Au niveau vocabulaire, chaque élément de l'arbre est appelé un nœud. Le nœud `html` contient deux enfants (`head` et `body`), chacun d'eux contenant également plusieurs enfants. Les éléments `head` et `body` sont frères, et leur parent est l'élément `html`. Notez que le nœud `meta`, les nœuds `text` et le

noeud `script` n'ont pas d'enfant. On dit alors que ces noeuds sont des *feuilles*. Ce vocabulaire est générique à tous les arbres informatiques.

Au niveau du DOM, il existe deux types de noeuds :

1. Les noeuds de type *élément html* qui sont des représentations des balises HTML (comme `div` ou `a`)
2. Les noeuds de type *textuels* qui ne sont pas dans des balises (le texte pur et les commentaires)

Un noeud de type textuel est donc forcément une feuille, puisqu'il ne peut pas avoir d'enfant.

Remarquez cependant qu'il est facile de transformer un noeud textuel en un élément html en l'encapsulant simplement dans une balise ``. L'intérêt ? Aucun, alors ne le faites pas 😊. Sauf dans des cas particuliers, bien sûr.

8.2.1 Accéder aux éléments html (old school)

JavaScript nous donne quelques méthodes pour sélectionner les éléments HTML du DOM. Nous allons voir les plus anciennes, parfaitement supportées par tous les navigateurs « normaux ».

Il est toutefois important de noter que si la première est une méthode du seul objet `document`, les autres sont des méthodes de tous les éléments HTML de la page.

8.2.1.1 `getElementById()`

Cette méthode de l'objet `document` permet de sélectionner un élément en fonction de son `id`, comme dans l'exemple suivant :

```
// On récupère la première div par son identifiant
const div1 = document.getElementById('div1');
alert(div1); // Seul le type est affiché
console.log(div1); // Seul le contenu est affiché
```

Le résultat est un pop-up contenant `[objectHTMLDivElement]`, ce qui prouve que nous avons bien récupéré un objet dont le type nous est donné et qui ma foi ressemble fort à un élément `div`. Remarquez également la différence d'affichage entre les méthodes `alert()` et `console.log()`. La seconde est bien plus verbeuse et nous affiche le contenu HTML de l'objet. Selon ce qu'on veut savoir (type ou contenu), on utilisera l'une ou l'autre.

8.2.1.2 `getElementsByTagName()`

Cette méthode existe pour tous les éléments HTML et permet de récupérer une collection de tous les éléments dont le nom de balise est passé en paramètre. Attention, du fait que c'est une collection, il peut y avoir plusieurs éléments, c'est pour ça qu'il y a un `s` à `Elements`.

Une collection est comme un tableau, en plus complexe. Pour nous, on considérera que c'est un tableau, car ça s'utilise comme un tableau :

```
// On récupère tous les liens contenus dans div1 dans un tableau
const links = div1.getElementsByTagName('a');
// On affiche tous les liens contenus dans le tableau
for (let link of links) console.log(link);
```

Ici nous avons récupéré tous les éléments `<a>` qui se trouvent dans l'élément d'`id` « `div1` ».

Comme dit en introduction, cette méthode est utilisable sur tous les éléments de la page HTML, permettant ainsi de sélectionner les éléments par nom de balise dans une sous-partie du document :

```
// On affiche toutes les div contenus dans body
let body = document.getElementById("body");
let divs = body. getElementsByTagName("div");
for (let div of divs.) console.log(div);
```

Remarquez qu'on peut facilement condenser l'écriture :

```
for (let div of
document.getElementById("body").getElementsByTagName("div"))
console.log(div);
```

Pour ma part, je ne trouve pas ça très lisible, mais parfois ça a du bon. Ça évite surtout de multiplier les variables. C'était utile avant ES6, maintenant beaucoup moins avec les mots-clés `let` et `const`. Si on lit la partie de droite (après le `of`), on a dans l'ordre :

- Avec l'objet `document`
- `getElementById("body")` : sélectionne l'élément dont l'id est `body` sur lequel on applique
- `getElementsByTagName("div")` : la sélection de tous les éléments `<div>` contenus dans l'élément courant (donc `body`).

Tous ces éléments sont parcourus (puisque une collection est itérable) dans la boucle `for...of` et affichés dans la console :

```
>> <div id="div1">...</div>
>> <div id="div2">...</div>
>> <div id="div3">...</div>
>>
```

8.2.1.3 `getElementsByClassName()`

Comme la méthode précédente, cette méthode s'applique à tout élément HTML. Elle retourne la collection des éléments HTML possédant la classe CSS indiquée en paramètre :

```
// On affiche tous les éléments ayant la classe "rouge"
for (let elt of
document.getElementById("body").getElementsByClassName("rouge"))
console.log(elt);
```

Le résultat est l'affichage de la div et du lien ayant la classe `rouge` :

```
>> <p id="p1" class="rouge">...</p>
>> <a id="a2" href="#" class="rouge">un second lien</a>
>>
```

8.2.1.4 `getElementsByName()`

Comme la méthode précédente, celle-ci retourne une collection d'objets sélectionnés par leur nom. Rappelons que depuis HTML5, l'utilisation de l'attribut `name` est dépréciée ailleurs que dans un formulaire. Cette méthode ne devrait donc être utilisée que sur des formulaires. D'autre part, cette méthode est dépréciée en XHTML, mais reste standard en HTML5. Pour combien de temps ?

8.2.2 Accéder aux éléments html (new school)

Plus récemment, deux méthodes sont apparues pour sélectionner les éléments du document. Ces méthodes permettent de cibler ces éléments grâce à leur sélecteur CSS, y compris CSS3 sauf, bien sûr, Internet Explorer 8 qui ne supporte que jusqu'à CSS2.1.

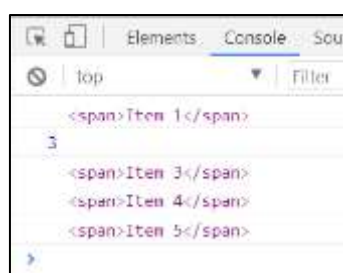
Ces deux méthodes sont `querySelector()` et `querySelectorAll()`. Elles prennent donc en paramètre une chaîne de caractères qui est le sélecteur CSS correspondant à ce que l'on cherche. La différence entre les deux est le nombre de résultats retournés. La première renvoie le premier élément trouvé, la seconde les renvoie tous sous forme de collection.

Ces méthodes sont fortement utiles, rappelez-vous-en, car si elles ne fonctionneront pas systématiquement sur tous les navigateurs (quoiqu'à l'heure actuelle ça ne devrait plus poser de problème), vous retrouverez leur pendant en `jQuery`, et celles-ci fonctionnent partout ! En particulier, Internet Explorer ne supporte pas les sélecteurs du CSS3 que vous pouvez trouver sur cette page : <https://www.w3.org/Style/css3-selectors-updates/WD-css3-selectors-20010126.fr.html#selectors>.

Exemple : Soit la page suivante, contenant un menu et un contenu :

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>DOM sélection CSS</title>
</head>
<body>
  <div id="menu">
    <div class="sub1">
      <span>Item 1</span>
      <span>Item 2</span>
    </div>
    <div class="sub2">
      <span>Item 3</span>
      <span>Item 4</span>
      <span>Item 5</span>
    </div>
  </div>
  <div id="content">
    <span>Le contenu de la page</span>
  </div>
  <script>
    const query = document.querySelector('#menu .sub1 span'),
          queryAll = document.querySelectorAll('#menu .sub2 span');
    console.log(query); // Affiche : "<span>Item 1<span>"
    console.log(queryAll.length); // Affiche "3"
    // Affiche : "<span>Item i<span>" pour i allant de 3 à 5
    for (let span of queryAll) console.log(span);
  </script>
</body>
</html>
```

Avec le script inclus dans cette page on a le résultat suivant :



8.2.3 Que choisir ?

Voici un petit tableau récapitulatif pour savoir quelle méthode utiliser pour cibler un élément :

Nombre d'éléments à obtenir	Critère de sélection	Méthode à utiliser
Plusieurs	Par balise	<code>getElementsByTagName</code>
Plusieurs	Par classe	<code>getElementsByClassName</code>
Plusieurs	Autre que par balise ou par classe	<code>querySelectorAll</code>
Un seul	Par identifiant	<code>getElementById</code>
Un seul (le premier)	Autre que par identifiant	<code>querySelector</code>

Théoriquement, on pourrait tout faire avec `querySelector()` et `querySelectorAll()`. Cependant ces deux méthodes souffrent de piètres performances par rapport aux autres méthodes. Il est donc important de toutes les connaître, du moins si la réactivité est importante pour vous.

8.2.4 Éditer les propriétés des éléments html

Revenons rapidement sur la notion d'héritage. Un élément html comme `<div>` est un objet `HTMLDivElement` comme nous avons pu le voir dans un `alert()`. Cet objet hérite de l'objet `HTMLElement`, qui lui-même hérite de `Element`, lui-même fils de `Node`. Toute cette chaîne d'héritage empile les propriétés (variables et méthodes) des objets parents dans le dernier héritier.

Rappelons également qu'un des fondements de la POO est l'encapsulation, qui interdit l'accès direct aux propriétés, mais qui propose des méthodes qu'on appelle *accesseurs* (*getter* et *setter*) pour les lire ou les modifier. Mais le modèle d'héritage par prototype de JS ne permet pas d'implémenter facilement cette encapsulation. De ce fait, les propriétés restent directement accessibles sans passer par les accesseurs. Cependant les accesseurs sont fournis, et il est préconisé de les utiliser car ils font un travail important : la mise en forme de l'information pour le *getter*, et la vérification de validité pour le *setter*. Je vous conseille donc d'utiliser les accesseurs en toute circonstance, même si on voit souvent l'accès direct aux propriétés utilisé dans les sources trouvées sur le web.

C'est l'objet `Element` qui va nous donner tout ce dont nous avons besoin (et par héritage tous ses descendants en profitent !).

8.2.4.1 Avec les accesseurs

L'objet `Element` nous donne deux méthodes pour accéder en lecture comme en écriture aux attributs des éléments d'une page : `getAttribute()` et `setAttribute()`. Le premier paramètre est le nom de l'attribut, le second dans le cas du *setter* est la nouvelle valeur à lui donner :

```
<body>
  <a id="unLien" href="http://www.google.fr/">Un lien modifié
  dynamiquement</a>
  <script>
    const lien = document.getElementById('unLien');
    const href = lien.getAttribute('href');
    console.log(href);
    // On change la cible du lien
    lien.setAttribute('href', 'http://www.ldnr.fr');
  </script>
</body>
```

On peut également vérifier si un élément possède un attribut avec la méthode `hasAttribute()` :


```
if (lien.hasAttribute('target')) console.log("Le lien possède l'attribut target");
else console.log("Le lien ne possède pas l'attribut target");
```

8.2.4.2 Accès direct aux attributs

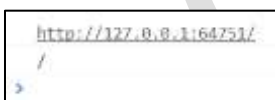
Dans les faits, comme il est difficile avec le prototypage d'encapsuler réellement les propriétés d'un objet, il est souvent possible d'accéder directement aux attributs des éléments. Ainsi le code précédent peut s'écrire :

```
const lien = document.getElementById('unLien'),
      href = lien.href; // Accès direct à l'attribut non encapsulé
console.log(href);
lien.href = 'http://www.ldnr.fr';
```

Vous voyez que les accesseurs ne sont plus utilisés. C'est souvent ainsi que l'on procède lorsqu'on travaille sur les formulaires, la syntaxe étant plus courte. Mais comme dit précédemment, attention : le fait d'ignorer les accesseurs peut changer grandement les choses.

Par exemple, testez le code suivant et constatez la différence entre les valeurs affichées :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Accesseurs vs attributs</title>
</head>
<body>
  <a href="/">Accueil</a>
  <script>
    const link = document.querySelector("a");
    console.log(link.href);
    console.log(link.getAttribute("href"));
  </script>
</body>
</html>
```



Hé oui ! La valeur de l'attribut est bien l'URL complète, alors que l'accesseur nous retourne seulement la racine du site !

8.2.4.3 Les exceptions à la règle

Rappelons que, comme dans tous les langages, on ne peut pas utiliser les mots-clés de JavaScript comme nom de variable dans le code source. Ainsi en va-t-il avec les mots `class` et `for`. Or ces mots sont aussi des noms d'attributs de certains éléments HTML : `class` est évidemment la classe CSS, et `for` est le champ de formulaire associé à un `label` qui permet de sélectionner le champ en cliquant sur le label. Ne pouvant utiliser ces noms, les propriétés s'appellent respectivement `className` et `htmlFor`.

Notez que si votre élément comporte plusieurs classes, c'est bien la chaîne de caractères complète qui est retournée en appelant la propriété `className`, il faudra donc la découper selon l'espace avec `split()` (voir les méthodes de la classe `String`). Sinon vous pouvez simplement utiliser la propriété `classList` qui contient une collection des classes de l'élément.

8.2.5 Gestion des contenus textuels

JavaScript propose deux propriétés pour connaître le contenu textuel d'un élément : `innerHTML` et `textContent`, cette dernière étant connue pour Internet Explorer sous le nom `innerText`. Cependant, IE 9 et plus connaissent également la version standardisée `textContent`.

La propriété `innerHTML` permet de récupérer le code HTML contenu dans l'élément sous forme de texte, y compris toutes les balises enfants de l'élément.

La propriété `textContent` (ou `innerText` pour Internet Explorer jusqu'à la version 8) fait comme `innerHTML` mais elle supprime toute trace de balise HTML.

En reprenant notre exemple initial :

```
<div id="div1">
  <p id="p1" class="rouge">Un paragraphe dans un div et <a id="a1"
href="#">un lien dans le paragraphe</a></p>
</div>
```

Avec le script suivant :

```
let div1 = document.getElementById('div1');
// Affichage du contenu textuel avec les balises
console.log(div1.innerHTML);
// affichage du contenu textuel sans les balises
console.log(div1.textContent);
```



8.3 Modifier la structure du DOM

8.3.1 Naviguer dans les nœuds du DOM

Une fois un nœud atteint par sélection, il est possible de naviguer à partir de ce nœud pour atteindre les parents, frères et fils du nœud courant. Il y a pour ce faire plusieurs méthodes proposées par JavaScript, que nous allons voir dans ce chapitre.

8.3.1.1 Les propriétés d'un nœud

La propriété `parentNode` permet d'accéder à l'élément html parent du nœud courant.

La propriété `nodeType` permet quant à elle de connaître le type du nœud. C'est un entier dont je vous donne ci-après la table de correspondance. Ce tableau étant bien sûr impossible à retenir, il est utile de savoir que chacune de ces valeurs est une constante de l'objet `Node`, que je vous fournis également dans le tableau :

Valeur	Type de nœud	Constante
1	Nœud élément	Node.ELEMENT_NODE
2	Nœud attribut	Node.ATTRIBUTE_NODE
3	Nœud texte	Node.TEXT_NODE

4	Nœud pour passage CDATA (relatif au XML)	Node.CDATA_SECTION_NODE
5	Nœud pour référence d'entité	Node.ENTITY_REFERENCE_NODE
6	Nœud pour entité	Node.ENTITY_NODE
7	Nœud pour instruction de traitement	Node.PROCESSING_INSTRUCTION_NODE
8	Nœud pour commentaire	Node.COMMENT_NODE
9	Nœud document	Node.DOCUMENT_NODE
10	Nœud type de document	Node.DOCUMENT_TYPE_NODE
11	Nœud de fragment de document	Node.DOCUMENT_FRAGMENT_NODE
12	Nœud pour notation	Node.NOTATION_NODE

- La propriété `nodeName` contient simplement le nom de l'élément. La valeur sera en minuscule en XML mais en HTML elle sera en majuscule. Pensez à la convertir si nécessaire. Pour les nœuds de type texte, la valeur retournée est `#text`.
- La propriété `nodeValue` contient le texte d'un nœud textuel. Cette propriété ne peut s'utiliser que sur un nœud de type `text`. Sur tout autre type de nœud, elle retourne la valeur `null`.

8.3.1.2 Les enfants

- Les propriétés `firstChild` et `lastChild` contiennent, comme leur nom l'indique, le premier et le dernier nœud enfant d'un nœud
- La propriété `childNodes` contient la collection des nœuds enfants d'un élément
- Les propriétés `firstElementChild` et `lastElementChild` contiennent le premier et le dernier élément html enfant du nœud courant
- La propriété `children` contient la collection des éléments html enfants d'un élément

Exemples : Dans la page suivante...

```
<div id="div1">
  <p id="p1" class="rouge">Un paragraphe dans un div et <a id="a1"
href="#">un lien dans le paragraphe</a></p>
</div>
```

... et avec le script suivant...

```
// Navigation entre les noeuds
const p = document.getElementById('p1');
console.log(p.firstChild); // texte " Un paragraphe dans un div et "
console.log(p.lastChild); // élément <a>
console.log(p.firstElementChild); // élément <a>
console.log(p.lastElementChild); // élément <a>
console.log(p.lastElementChild.nodeValue); // null
console.log(p.lastChild.firstChild.nodeValue); // valeur du texte "un lien
dans le paragraphe"
console.log(p.lastChild.firstChild); // texte "un lien dans le paragraphe"
// Utilisation de la collection childNodes de tous les noeuds enfants
for (let node of document.getElementById('p1').childNodes)
console.log(node);
// Utilisation de la collection children de tous les éléments html enfants
for (let elt of document.getElementById('p1').children) console.log(elt);
```

... on obtient le résultat suivant :

```
"Un paragraphe dans un div et "  
<a id="a1" href="#">un lien dans le paragraphe</a>  
<a id="a1" href="#">un lien dans le paragraphe</a>  
<a id="a1" href="#">un lien dans le paragraphe</a>  
null  
un lien dans le paragraphe  
"un lien dans le paragraphe"  
"Un paragraphe dans un div et "  
<a id="a1" href="#">un lien dans le paragraphe</a>  
<a id="a1" href="#">un lien dans le paragraphe</a>
```

Remarquez que pour accéder au texte contenu dans l'élément `<a>` il a fallu sélectionner d'abord son premier fils, qui est la feuille texte dont le contenu est le texte lui-même. Comparez les résultats dans la console pour mieux comprendre.

Constatez également la différence entre les propriétés `childNodes` qui contient deux nœuds et `children` qui ne contient qu'un seul élément html.

8.3.1.3 Les frères

Il est possible également de cibler un élément frère de l'élément courant dans le DOM avec les propriétés `nextSibling` et `previousSibling`. De la même façon que pour les enfants, il est possible d'ignorer les nœuds textuels avec les propriétés `nextElementSibling` et `previousElementSibling`.

8.3.1.4 Pourquoi tant de haine ?

Toutes ces propriétés, on s'y perd ! Pourquoi ces différences subtiles entre choisir tous les nœuds et seulement les éléments html ? La réponse se trouve dans l'exemple suivant : si on compare les deux codes suivants, il n'y a *a priori* aucune différence :

```
<div>  
  <p>Item 1</p>  
  <p>Item 2</p>  
  <p>Item 3</p>  
</div>
```

Et :

```
<div><p>Item 1</p><p>Item 2</p><p>Item 3</p></div>
```

Et pourtant si ! Niveau interprétation HTML, c'est la même chose, incontestablement. Mais voilà, dans le premier code, les retours à la ligne sont considérés comme des nœuds textuels, tout comme le seraient des espaces. Le DOM du premier contient donc 7 enfants (trois nœuds `<p>` et quatre nœuds textuels vides), alors que le second n'en contient que trois (les trois éléments `<p>`). De l'intérêt des propriétés ciblant les éléments html par rapport aux propriétés ciblant tous les nœuds sans distinction. Ça permet d'écrire du code HTML propre sans pour autant impacter la recherche des éléments importants en JS.

8.3.2 Ajouter des éléments au DOM

JavaScript, et c'est bien là sa force, permet de modifier en profondeur le DOM en y ajoutant et supprimant des éléments à l'envi. Nous allons voir dans cette partie comment ajouter des éléments. Ceci se fait en trois étapes :

1. Créer l'élément à ajouter.
2. Hypothétiquement lui affecter des attributs.
3. L'insérer dans le DOM.

Ce n'est qu'à l'issue de la troisième étape que l'élément existera dans le DOM, modifiant de fait la page HTML immédiatement.

8.3.2.1 Créer un nœud

Créer un élément html se fait simplement avec la méthode `createElement()` de l'objet `document`. Cette méthode prend en paramètre le type de l'élément qu'on souhaite créer. Ainsi, pour créer un élément de type lien, nous utiliserons la syntaxe suivante :

```
const newLink = document.createElement('a');
```

Créer un nœud textuel se fait avec la méthode `createTextNode()`, qui prend simplement en paramètre le contenu du nœud textuel :

```
const newTextNode = document.createTextNode("click and see the best site ever!");
```

Cela dit, on peut tout aussi bien utiliser les propriétés `innerHTML` ou `textContent` pour ajouter du texte dans un élément, rappelez-vous-en.

8.3.2.2 Affecter des attributs

Notre élément existe, on y accède donc comme tout autre élément, soit avec les accesseurs `setAttribute()` et `getAttribute()` soit directement avec les propriétés :

```
newLink.setAttribute('id', 'myLink');
newLink.title = "The best site ever!";
newLink.href = 'https://www.ldnr.fr';
```

8.3.2.3 Insérer un nœud dans le DOM

On peut utiliser deux possibilités pour ajouter un nœud : `appendChild()` et `insertBefore()`.

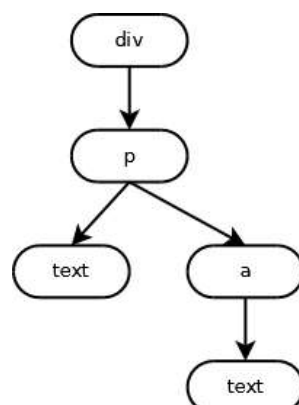
La méthode `appendChild(newElmt)`, qui prend en paramètre le nœud à ajouter (ici `newElmt`), permet d'ajouter ce nœud en tant que dernier enfant du nœud actuel.

La méthode `insertBefore(newElmt, refElmt)` prend deux paramètres. Le premier (ici `newElmt`) est le nœud à ajouter, le second (ici `refElmt`) est le nœud enfant avant lequel `newElmt` sera ajouté. Si `refElmt` est null, `newElmt` sera inséré en tant que dernier enfant, émulant ainsi le fonctionnement de `appendChild()`.

Exemple : Partons de notre code HTML de base :

```
<div id="div1">
  <p id="p1" class="rouge">Un paragraphe dans un div et <a id="a1"
href="#">un lien dans le paragraphe</a></p>
</div>
```

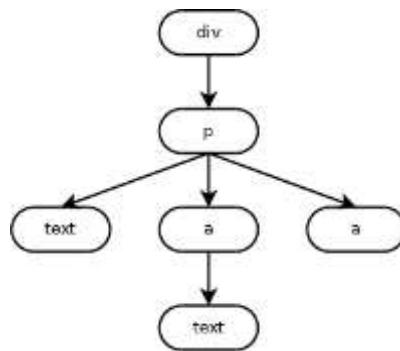
Notre DOM pour cette partie est :



Avec le code suivant :

```
const p1 = document.getElementById('p1'),
      newLink = document.createElement('a');
newLink.setAttribute('id', 'myLink');
newLink.title = "The best site ever!";
newLink.href = 'http://www.ldnr.fr';
p1.appendChild(newLink);
```

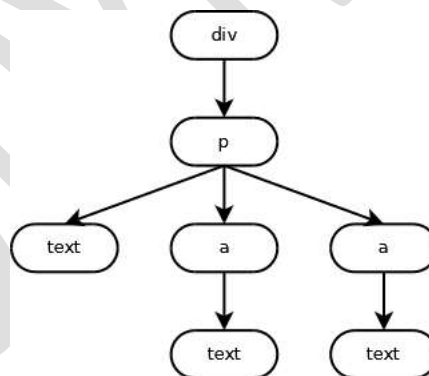
Notre DOM devient :



Il manque quelque chose, n'est-ce pas ? Il manque du texte dans notre lien, simplement... Il faut donc créer un nœud textuel fils du nouvel élément. Et ça, on sait faire :

```
const newTextNode = document.createTextNode("click and see the best site ever!");
newLink.appendChild(newTextNode);
```

Exécutez ça, vous verrez apparaître notre lien puisque le DOM devient :



En revanche ce n'est pas très propre. On crée un élément qu'on insère dans le DOM alors qu'il est encore incomplet. Il vaut mieux créer toute l'arborescence de nœuds que l'on souhaite insérer, et l'insérer une bonne fois pour toute.

Ainsi, et si pour changer je veux insérer mon lien avant l'autre lien, je vais utiliser le code suivant :

```
const p = document.getElementById('p1'),
      newLink = document.createElement('a');
// Construction de mon arbre de nodes
newLink.setAttribute('id', 'myLink');
newLink.title = "The best site ever!";
newLink.href = 'http://www.ldnr.fr';
newLink.textContent = "click and see the best site ever!";
// Insertion au bon endroit
```

```
p.insertBefore(newLink, p.firstChild);
```

Notez l'utilisation obligatoire ici de la propriété `firstElementChild`, et de `textContent` au lieu de créer un nœud textuel.

8.3.3 Dupliquer un nœud

Nous avons vu que ce qui est réellement stocké dans une variable n'est pas l'objet lui-même mais l'adresse en mémoire de l'objet, ce qu'on appelle sa référence. Comme une carte au trésor qui permet de trouver l'objet dans la mémoire.

Lorsqu'on copie une variable dans une autre, ce qu'on copie est le contenu de la boîte dans la mémoire, c'est-à-dire l'adresse de l'objet, ou encore la carte au trésor pour accéder à l'objet. Ce qui veut dire que la deuxième variable contient la même carte au trésor, et pointe donc le même objet (le même appartement pour poursuivre notre analogie). Donc si je modifie l'objet pointé par la deuxième variable, en fait je modifie l'objet pointé aussi par la première variable.

Exemple :

```
const s1 = document.body.appendChild(document.createElement('span'));
s1.textContent = 'Du texte en plus !';
const s2 = s1; // on copie la référence !
s2.textContent = 'Un autre texte'; // on modifie via la seconde référence
console.log(s1.textContent); // retourne 'Un autre texte' !
```

Explications : dans la première ligne on crée un nouvel élément `` qu'on ajoute au `<body>` (le tout en une seule ligne, c'est bien aussi pour ça l'objet). La référence à cet élément est stockée dans la variable `s1`, car les méthodes `appendChild()` ou `insertBefore()` retournent la référence de l'objet inséré. À la deuxième ligne on met du contenu dans `s1`, donc on modifie l'appartement pointé par `s1`. À la troisième ligne on copie `s1` dans `s2`. Mais ce qui est copié est bien la référence à notre élément. Ainsi, à la quatrième ligne, lorsqu'on modifie le contenu grâce à `s2`, on modifie l'appartement pointé par `s2`, qui est le même que celui pointé par `s1` ! Ce qu'on vérifie en affichant le contenu de `s1` à la dernière ligne.

Donc si je veux dupliquer un nœud, je dois m'y prendre autrement. Il ne s'agit pas de simplement copier son adresse, mais bien de créer un clone. JS met à notre disposition la méthode `cloneNode()` à cette fin. Cette méthode ne prend qu'un seul paramètre booléen optionnel, qui indique si on veut (`true`) ou si on ne veut pas (`false`) dupliquer également les attributs du nœud et toute sa descendance (toute l'arborescence des enfants).

```
const span1 = document.createElement('span');
span1.textContent = 'Un span à dupliquer';
const span2 = span1.cloneNode(false); // contenu textuel non dupliqué
document.body.appendChild(span1); // affiche le contenu
document.body.appendChild(span2); // crée le span sans contenu
```

Attention : le paramètre est optionnel, MAIS le comportement par défaut a changé au cours des évolutions des versions du DOM. Donc, pour éviter toute surprise, indiquez systématiquement ce paramètre.

Attention bis : si vous positionnez le paramètre à `true`, vous risquez de dupliquer l'`id` s'il existe, ce qui, rappelons-le, est formellement interdit. Pensez donc à bien le modifier par la suite, avant d'insérer l'élément dans le DOM...

Attention : même avec le paramètre à `true`, les événements ajoutés par `addEventListener()` ou `onXXX()` ne sont pas dupliqués. Nous verrons la gestion des événements dans la suite du cours, nous en reparlerons à cette occasion.

8.3.4 Remplacer un élément par un autre

Pour remplacer un élément par un autre (préalablement construit, bien sûr), il faut utiliser la méthode `replaceChild()`. Cette méthode prend deux paramètres. Le premier est le nouvel élément, le second l'élément à remplacer.

Exemple : Si on garde notre page web de référence :

```
<div id="div1">
  <p id="p1" class="rouge">Un paragraphe dans un div et <a id="a1"
href="#">un lien dans le paragraphe</a></p>
</div>
```

Avec le code suivant on change le contenu du texte du lien :

```
const lien = document.querySelector('a');
const newText = document.createTextNode('et un hyperlien');
lien.replaceChild(newText, lien.firstChild); // On remplace le node text
```

8.3.5 Supprimer un élément

Pour supprimer un élément, il suffit d'appliquer la méthode `removeChild()` sur son parent, en passant en paramètre l'élément à supprimer. Notez que cette méthode retourne l'élément supprimé, il est donc tout à fait possible de l'insérer par la suite dans un autre endroit du DOM.

Exemple : Dans la page web de référence, pour déplacer le lien avant le paragraphe, on peut faire :

```
const div1 = document.getElementById('div1'),
    lien = document.querySelector('a'),
    p = document.getElementById('p1');
const old = lien.parentNode.removeChild(lien);
div1.insertBefore(old, p);
```

8.3.6 C'est du brutal !

Toutes ces méthodes que nous avons vues permettent de cibler de façon pointue les éléments de notre DOM. Mais n'oublions pas qu'on peut aussi directement modifier le contenu d'un élément html avec ses propriétés `innerHTML` et `textContent` qui permettent donc d'écrire du code HTML directement.

Ce n'est évidemment pas très élégant, mais parfois bien plus rapide...

À ce stade du cours, je ne peux pas passer sous silence la méthode `insertAdjacentHTML()` de la classe `Element`. Cette méthode prend deux paramètres : le second est un texte représentant un code HTML, et le premier une position de référence. La méthode parcourt le texte, y trouve les nœuds correspondants et les insère à la position précisée par le premier paramètre. Cette méthode est beaucoup plus rapide que d'utiliser `innerHTML` parce qu'elle ne parcourt pas l'élément sur lequel elle est utilisée.

Le premier paramètre peut être :

- `'beforebegin'` : avant l'élément lui-même
- `'afterbegin'` : juste au début de l'élément, avant son premier enfant
- `'beforeend'` : juste à la fin de l'élément, après son dernier enfant

- 'afterend' : après l'élément lui-même

Donc si on représente les positions par rapport à un élément `div` par exemple, on a :

```
beforebegin
<div>
  afterbegin
  premier enfant de la div
  ...
  dernier enfant de la div
  beforeend
</div>
afterend
```

Exemple : soit le code HTML suivant :

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Méthode InsertAdjacentHtml</title>
</head>
<body>
  <div id="menu">
    <ul id="list">
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </div>
  <script>
    const sub1 = document.getElementById("list");
    // Ajout d'un élément avant la liste
    sub1.insertAdjacentHTML("beforebegin", "<div>Une première
div</div>");
    // Ajout d'un élément au début de la liste
    sub1.insertAdjacentHTML("afterbegin", "<li>Item 1</li>");
    // Ajout d'un élément à la fin de la liste
    sub1.insertAdjacentHTML("beforeend", "<li>Item 4</li>");
    // Ajout d'un élément après la liste
    sub1.insertAdjacentHTML("afterend", "<div>Une dernière div</div>");
  </script>
</body>
</html>
```

Nous obtenons ce résultat sur la page HTML :



9 Modifier le CSS

9.1 Obtenir le style d'un élément

De même qu'on peut obtenir tout attribut d'un élément html dans le DOM, on peut également obtenir l'attribut `style`. Vous le savez, cet attribut permet de définir le style au sens mise en forme CSS de l'élément. On peut donc faire ce qu'on veut en réécrivant complètement le style directement dans l'élément.

Mais si l'élément est mis en forme uniquement par CSS extérieur, l'attribut `style` ne contient aucune des valeurs définies dans le CSS. En effet, cet élément ne contient que les valeurs par défaut modifiées par l'attribut `style` situé dans la balise.

Pour obtenir la mise en forme finale, après application des CSS, il faut utiliser la fonction `getComputedStyle()` (méthode de l'objet `window`) qui prend l'élément en paramètre. Cette fonction retourne un objet de type `CSSStyleDeclaration` contenant toutes les règles CSS du style. Il est alors possible de consulter les valeurs des différents attributs de style.

9.2 Modifier le style d'un élément

C'est bien simple, il suffit de modifier la propriété `style` de l'élément. Il n'y a pas d'autre solution, on ne peut en effet pas atteindre les fichiers CSS avec JavaScript.

Exemple :

```
p.style.color = "red";
p.style.margin = "50px";
```

En revanche, pour les propriétés CSS qui contiennent un underscore, la nomenclature change légèrement. Il convient de supprimer l'underscore et de mettre une majuscule au mot suivant (notation camelCase) :

```
p.style.fontFamily = "Arial";
p.style.backgroundColor = "black";
```

Notez que si vous voulez appliquer un style à tous les éléments d'une même balise (par exemple tous les éléments `li`), il faut en passer par une boucle :

```
// Modification de la couleur de tous les li
const listItems = document.getElementsByTagName("li");
for (let li of listItems) li.style.color = "red";
```

Mais le plus correct reste encore de prévoir dans le CSS des classes qui permettront d'obtenir le même résultat, et d'ajouter ou supprimer ces classes dans l'élément HTML grâce à la gestion de l'attribut `classList`, avec les méthodes `add()` et `remove()`. Ainsi, on ne mélange pas le fond et la forme, et on n'ajoute pas de style dans les balises directement.

10 La gestion événementielle

10.1 Qu'est-ce qu'un évènement ?

Un évènement est une action produite dans la page web, par exemple le survol d'un élément par la souris. JavaScript permet de surveiller certains évènements, et de réagir lorsque ces évènements se produisent, en appelant une fonction définie par le codeur.

10.2 Rappel sur le focus

On dit qu'un élément obtient le focus lorsqu'il devient sélectionné. Pas seulement par un click, mais aussi par navigation entre les différents champs ou liens avec la touche `tab` par exemple.

Lorsqu'un élément perd le focus (quelle qu'en soit la cause), il déclenche l'évènement `blur`.

10.3 Écouteurs

Pour qu'un évènement déclenche une action, il est nécessaire d'y ajouter un écouteur (en anglais : listener) -on parle aussi de gestionnaire d'évènement mais c'est plus long. Cet écouteur déclenchera automatiquement l'appel à une fonction dès que l'évènement survient. Cette fonction est appelée fonction de rappel ou, plus communément en français, une fonction callback souvent abrégé en callback tout court. C'est donc bien l'apparition de l'évènement qui déclenche le callback.

Pour ajouter un écouteur à un évènement, il existe plusieurs façons, selon que l'on veut se conformer au DOM-0 ou au DOM-2. Je présenterai plutôt la façon « récente » en parlant succinctement de la façon « ancienne » au cas où vous tomberiez dessus dans de vieux scripts (ou des scripts récents écrits par de vieux codeurs !).

10.3.1 Écouteur (new school)

Pour se conformer au DOM-2, il convient d'utiliser la méthode `addEventListener(event, callback, useCapture)` sur l'élément du DOM à écouter. Cette méthode prend donc trois paramètres :

1. Premier paramètre : `event` est une chaîne représentant l'évènement à surveiller (voir tableau ci-dessous)
2. Second paramètre : `callback` est l'objet qui recevra une notification lorsque l'évènement se produit. Pour nous ce sera le callback, donc une simple fonction.
3. Troisième paramètre(optionnel) : `useCapture` est un booléen indiquant le sens de propagation de l'évènement. Nous reviendrons plus tard sur ce paramètre que vous pouvez aisément oublier.

On peut ajouter autant de callback qu'on veut pour un même élément en ajoutant autant d'écouteurs, c'est une des grandes forces de cette façon de faire.

Pour travailler sur les évènements, nous allons changer de page web de travail. Je vous propose celle-ci :

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Gestion des évènements</title>
</head>
<body>
```

```
<button id="btn">Cliquez-moi !</button>
<script src="../../js/evenements.js"></script>
</body>
</html>
```

Elle est d'une puissance incomparable puisqu'elle ne possède qu'un bouton sur lequel est écrit : "Cliquez-moi !".

Voici un premier écouteur :

```
function afficherClic() {
    console.log('Clic !');
}
function afficherBouh() {
    console.log('Bouh !');
}
const bouton = document.getElementById('btn');
bouton.addEventListener('click', afficherClic);
bouton.addEventListener('click', afficherBouh);
```

Explications :

- On commence par définir une fonction `afficherClic()` qui affiche *Clic !* dans la console et une autre fonction `afficherBouh()` qui affiche *Bouh !* dans la console.
- Puis on sélectionne notre bouton grâce à son `id`.
- Enfin on ajoute deux écouteurs sur ce bouton pour l'évènement `click`. Ainsi, à chaque fois qu'on va cliquer sur le bouton, l'évènement associé (qui est écouté par ce bouton) déclenchera l'appel des callbacks, à savoir les fonctions `afficherClic()` et `afficherBouh()`.

Si vous essayez ce code, à chaque fois que vous cliquez sur le bouton deux nouvelles lignes apparaissent dans la console.

Remarquez que le callback est écrit sans parenthèse. Le paramètre attendu est effectivement le nom de la fonction, il ne s'agit nullement de l'appel à la fonction (ce qui se produirait si on mettait les parenthèses). C'est le gestionnaire d'évènement qui se chargera d'appeler cette fonction lorsque l'évènement surviendra. C'est une erreur courante qui empêche le gestionnaire d'évènement de fonctionner correctement.

De la même façon qu'on peut ajouter un écouteur, on peut le supprimer avec la méthode `removeEventListener(event, callback)` de l'élément.

```
bouton.removeEventListener('click', afficherClic);
```

Après l'ajout de cette ligne, la console n'affichera plus qu'un seul message à chaque clic. Remarquez au passage que lorsqu'un message se répète `n` fois dans la console, plutôt que de l'afficher `n` fois, il n'est affiché qu'une seule fois avec un compteur devant. C'est donc ce compteur qui va s'incrémenter à chaque clic.

10.3.2 Écouteur (old school)

La méthode ancienne pour mettre un écouteur est plus limitée. Il s'agit des fameux attributs `onXXX` où `xxx` est l'évènement. On trouve donc `onclick`, `onkeyup`, `onfocus`, etc... qu'on peut directement ajouter dans le code HTML, au sein de la balise :

```
<a href="http://www.ldnr.fr" id="clickme" onclick="console.log('Vous avez cliqué !');">Cliquez-moi !</a>
```

On peut directement mettre du code JavaScript ou appeler une fonction définie par ailleurs.

Ces évènements sont, bien sûr, tout aussi facilement écoutables directement dans le code JavaScript. Pour assigner un écouteur à un élément du DOM nous allons passer par les propriétés de l'élément. Ces propriétés sont le reflet des attributs de la balise HTML : `onXXX`. À ces propriétés nous allons associer directement le callback :

```
let elt = document.getElementById('clickme');
elt.onclick = function() {
    console.log("Vous avez cliqué !");
};
```

Nous voyons immédiatement les avantages de la méthode `addEventListener()` sur les propriétés :

Il est possible d'assigner plusieurs callbacks simplement, ainsi que de les supprimer avec `removeEventListener()`. Avec les propriétés, c'est plus complexe : il faut un callback qui appelle lui-même un ensemble de fonctions. On peut imaginer utiliser un tableau de fonctions et gérer ainsi le tableau pour émuler les possibilités du DOM-2, mais ça reste incontestablement moins accessible.

Elle donne un contrôle plus fin sur les phases d'activation de l'écouteur (capture ou bouillonnement - voir plus loin)

Elle fonctionne sur tous les nœuds du DOM, pas que sur les éléments HTML.

Nous nous cantonnerons donc aux prérogatives du DOM-2 dans la suite de ce cours.

10.3.3 Internet Explorer

Bien sûr, Internet Explorer ne fonctionne pas comme tous les autres, sauf depuis sa version 11. Ainsi, au lieu d'utiliser `addEventListener()` et `removeEventListener()` il conviendra d'utiliser plutôt les méthodes `attachEvent()` et `detachEvent()`.

Ces deux méthodes fonctionnent comme celles que nous connaissons à quelques différences près :

- Il n'y pas de troisième paramètre (puisque seule la phase de bouillonnement est supportée).
- Le type d'évènement (premier paramètre) est préfixé par `on` (par exemple `onfocus`).
- Il n'y a pas d'équivalent à `this` ou `currentTarget`, ce qui peut parfois être bien gênant, la cible n'étant pas systématiquement le même nœud que celui auquel on a attaché l'écouteur.

Donc si vous voulez un code « universel », il faudra ruser :

```
if (elt.addEventListener){
    elt.addEventListener('click', callback, false);
} else if (elt.attachEvent) {
    elt.attachEvent('onclick', callback);
}
```

10.4 Les différents évènements

Il y a une énorme quantité d'évènements possible, je vous convie à en lire la liste sur la documentation officielle : <https://developer.mozilla.org/en-US/docs/Web/Events>.

Il est bien évident que nous n'allons pas tout voir, ce serait impossible. Cependant nous allons voir les essentiels, en apprenant doucement comment se servir de ces évènements.

Voici les évènements essentiels à connaître :

Catégorie	Nom de l'évènement	Action pour le déclencher
Souris	click	Cliquer (appuyer puis relâcher) sur l'élément
Souris	dblclick	Double-cliquer sur l'élément
Souris	mouseover	Faire entrer le curseur sur l'élément
Souris	mouseout	Faire sortir le curseur de l'élément
Souris	mousedown	Appuyer (sans relâcher) sur n'importe quel bouton de la souris sur l'élément
Souris	mouseup	Relâcher n'importe quel bouton de la souris sur l'élément
Souris	mousemove	Faire déplacer le curseur sur l'élément
Clavier	keydown	Appuyer (sans relâcher) sur une touche de clavier sur l'élément
Clavier	keyup	Relâcher une touche de clavier sur l'élément
Clavier	keypress	Frapper (appuyer puis relâcher) une touche de clavier sur l'élément sauf les touches <i>Shift</i> , <i>Caps Lock</i> et <i>Fn</i>
Formulaire	focus	Cibler l'élément
Formulaire	blur	Annuler le ciblage de l'élément
Formulaire	change	Changer la valeur d'un élément spécifique aux formulaires (<code>input</code> , <code>checkbox</code> , etc.)
Formulaire	input	Taper un caractère dans un champ de texte <code>input</code> ou <code>textarea</code> . Pour les <code>input</code> de type <code>radio</code> ou <code>checkbox</code> , cet évènement n'est pas déclenché en cliquant sur un contrôle car la valeur de l'attribut <code>value</code> ne change pas
Formulaire	select	Sélectionner le contenu d'un champ de texte (<code>input</code> , <code>textarea</code> , etc.)
Formulaire	submit	Envoyer le formulaire
Formulaire	reset	Réinitialiser le formulaire
Page	load	La page est entièrement chargée
Page	beforeUnload	Fermeture de la page web (fermeture du navigateur ou de l'onglet)

Nous allons voir dans un premier temps les évènements afférents aux divers périphériques. Les évènements liés aux formulaires seront vus à part dans le chapitre suivant.

Chaque évènement déclenché s'accompagne de la création d'un objet `Event` qui est systématiquement passé en paramètre au callback. Que vous l'utilisiez ou pas ne dépend que de vous ! Je vous convie à jeter un œil attentif à ce qu'on peut faire avec un tel objet sur la documentation officielle : <https://developer.mozilla.org/en-US/docs/Web/API/Event>.

En particulier on trouve les propriétés `type` et `target`, qui donnent bien évidemment le type et la cible de l'évènement, autrement dit l'élément du DOM auquel il est destiné.

Voici un exemple de callback utilisant l'objet `Event` :

```
btn.addEventListener('click', function (evt) {
    console.log("Evènement : " + evt.type + ", texte de la cible : " +
    evt.target.textContent);
});
```

Vous remarquerez qu'on a ici utilisé une fonction anonyme. Pour utiliser une fonction nommée, ça ne change pas grand-chose :

```
function afficherMessage(e) {
    console.log("Evènement : " + e.type + ", texte de la cible : " +
    e.target.textContent);
}
bouton.addEventListener('click', afficherMessage);
```

10.4.1 Les évènements clavier

Le plus courant est d'utiliser l'évènement `keypress` directement sur la page elle-même. On peut ensuite utiliser l'évènement pour obtenir plein d'informations sur sa source.

Essayez le code suivant :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Gestion évènementielle du clavier</title>
</head>
<body>
    <script>
        document.addEventListener("keypress", appui);
        function appui(evt) {
            console.log(`Vous avez appuyé sur la touche ${evt.key} qui a
pour code ASCII ${evt.charCode}`);
        }
    </script>
</body>
</html>
```

Dans cet exemple j'utilise les valeurs des propriétés `key` et `charCode` de l'évènement `evt` passé en argument de mon callback.

Vous remarquerez toutefois que l'évènement `keypress` ne détecte pas l'appui sur les touches spéciales (*SHIFT*, *CTRL*, *ALT*, ...). Pour l'appui et le relâchement de toutes les touches, on a les évènements `keydown` et `keyup` respectivement. Dans l'exemple précédent, il suffit de remplacer la partie script par celle-ci :

```
<script>
    // Gestion de l'utilisation d'une touche "standard"
    document.addEventListener("keypress", appui);
    // Gestion de l'appui sur une touche quelconque
    document.addEventListener("keydown", infosTouche);
```

```
// Gestion du relâchement d'une touche quelconque
document.addEventListener("keyup", infosTouche);
function appui(evt) {
    console.log(`Vous avez appuyé sur la touche ${evt.key} qui a
pour code ASCII ${evt.charCode}`);
}
function infosTouche(evt) {
    console.log(`Evènement clavier : ${evt.type}, touche :
${evt.keyCode}`);
}
</script>
```

10.4.2 Les évènements souris

Le clic sur un élément du DOM provoque un évènement de type `click`. Si on l'écoute, on peut en récupérer de nombreuses informations via ses différentes propriétés. Nous trouvons parmi elles les suivantes :

- `button` : permet de connaître quel bouton a été utilisé. Cela dit, la majorité des navigateurs désactivent les clics droits et centre pour les assigner à d'autres tâches. Donc d'une manière générale, la valeur retournée sera toujours gauche...
- `clientX` et `clientY` sont les coordonnées horizontale et verticale (respectivement) du pointeur au moment du clic. Ces coordonnées sont relatives à la zone de la page dans laquelle on clique (la zone visualisée dans le navigateur, le viewport). Rappelons que le 0 se trouve en haut à gauche de l'écran, et que les X sont positifs vers la droite et les Y vers le bas.
- `pageX` et `pageY` sont les coordonnées absolues dans la page en X et Y de la souris. Elles sont différentes des précédentes si la page est scrollable.

De la même façon que pour les appuis sur les touches du clavier, nous avons la gestion de l'appui/relâchement du bouton de souris avec les évènements `mousedown` et `mouseup`. Il est à noter que ces deux évènements détectent les trois boutons, contrairement à l'évènement `click`.

Exemple : Code HTML :

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Gestion des évènements souris</title>
</head>
<body>
    <div style="border: 1px solid red; height: 10000px">
        Cliquez, scrollez, et recliquez !
    </div>
    <script>
        // Gestion évènement appui sur un bouton de souris
        document.addEventListener("mousedown", infoSouris);
        // Gestion évènement relâchement d'un bouton de souris
        document.addEventListener("mouseup", infoSouris);
        // Gestion évènement clic
        document.addEventListener('click', infoSouris);

        // Fonction donnant des informations sur la position de la souris
        // et du bouton utilisé
        function infoSouris(event) {
            const x1 = event.clientX,
```

```

        y1 = event.clientY,
        x2 = event.pageX,
        y2 = event.pageY,
        boutonSouris = getBoutonSouris(event.button);
        console.log(`Évènement souris : ${event.type} avec le bouton
        ${boutonSouris}
        clientX coords: X: ${x1}, Y: ${y1}
        pageX coords: X:${x2}, Y: ${y2}`);
    }
    // Fonction déterminant le bouton utilisé par l'évènement souris
    function getBoutonSouris(code) {
        let bouton = "inconnu";
        switch (code) {
            case 0: // 0 est le code du bouton gauche
                bouton = "gauche";
                break;
            case 1: // 1 est le code du bouton du milieu
                bouton = "milieu";
                break;
            case 2: // 2 est le code du bouton droit
                bouton = "droit";
                break;
        }
        return bouton;
    }
}
</script>
</body>
</html>

```

10.4.3 Les évènements page

Il y a principalement deux évènements pour la page : `load` et `beforeunload`. Le premier est déclenché lorsque la page est complètement chargée, le second lorsqu'elle est fermée, que ce soit parce qu'on navigue vers une autre page, qu'on ferme l'onglet ou qu'on ferme le navigateur.

Ces deux évènements s'attachent bien évidemment à l'objet `window`.

Le premier évènement est fondamental lorsqu'on veut interagir directement avec la page au démarrage. Toute action menée par JavaScript doit s'assurer que la page est bien complètement chargée, sinon on risque des surprises :

```

window.addEventListener("load", function () {
    console.log("Page entièrement chargée");
});

```

Normalement et de façon standardisée, le second évènement provoque une demande de confirmation si la valeur de la propriété `returnValue` de l'objet `Event` est modifiée. Mais certains navigateurs s'appuient plutôt sur la valeur de retour du callback. Pour être sûr de fonctionner partout, il faut utiliser ce genre de code :

```

window.addEventListener("beforeunload", function (e) {
    const message = "Voulez-vous vraiment partir ?";
    e.returnValue = message; // Provoque une demande de confirmation
    (standard)
    return message; // Provoque une demande de confirmation (certains
    navigateurs)
});

```


Notez que les navigateurs récents ne déclenchent pas l'évènement `beforeunload` si l'utilisateur n'a pas interagit avec la page avant de la fermer. Au contraire, ils provoquent une erreur `Blocked attempt to show a 'beforeunload' confirmation panel for a frame that never had a user gesture since its load`. Vous pouvez voir cette erreur en gardant l'historique de l'affichage de la console.

10.5 Capture et bouillonnement

Vous pouvez aisément vous passer de cette partie du cours, elle est juste là pour vous permettre d'aller un peu plus loin.

Lors du déclenchement d'un évènement sur un élément du DOM, il se produit un certain nombre de choses :

- L'évènement (objet `Event`) est créé.
- Phase de capture : l'évènement se propage de la racine du document (incluse) jusqu'à la cible (exclue).
- L'évènement atteint la cible.
- Phase de bouillonnement (en anglais `bubbling`) : l'évènement se propage dans le sens inverse, à savoir de la cible (exclue) jusqu'à la racine du document (incluse).

Pour intercepter l'évènement, et donc pouvoir appeler le callback, il faut deux choses :

- Un type d'évènement.
- Une phase : capture ou bouillonnement, sachant qu'on inclut la cible dans la phase de bouillonnement mais pas dans la phase de capture.

Pour faire court, le callback n'est appelé que lorsque l'évènement s'est propagé jusqu'à la cible si on choisit la phase de capture, et tout de suite si on choisit la phase de bouillonnement. Le choix se fait avec le troisième paramètre de la méthode `addEventListener()` : la valeur `true` signifie la phase de capture, la valeur `false` indique la phase de bouillonnement. La valeur est `false` par défaut. Autrement dit, par défaut, l'appel au callback est immédiat lors de la survenance de l'évènement.

Il est important de savoir que les navigateurs font un peu n'importe quoi sur ces notions. Internet Explorer ne supporte que la phase de bouillonnement, alors que d'autres navigateurs (dont Firefox) incluent la cible dans la phase de capture. Aussi il n'est pas inutile d'ignorer pour l'instant ce paramètre. Mais si vous voulez vous convaincre de son fonctionnement, essayez ce code :

```
<body>
  <p id="para">Un paragraphe avec un
    <button id="btn">bouton</button> à l'intérieur
  </p>

  <script>
document.addEventListener("click", function () {
  console.log("document");
}, false);
// Gestion du clic sur le paragraphe
document.getElementById("para").addEventListener("click", function () {
  console.log("paragraphe");
}, false);
// Gestion du clic sur le bouton
document.getElementById("btn").addEventListener("click", function (e) {
  console.log("bouton");
});
  </script>
</body>
```

```
}, false);  
</script>  
</body>
```

Lorsque vous cliquez sur le bouton, vous obtenez l’affichage normal de la propagation par bouillonnement (de l’évènement vers l’ancêtre le plus lointain, à savoir `document`). Maintenant si vous changez les trois valeurs booléennes pour `true`, vous obtenez l’affichage inverse pour la propagation par capture.

Amusez-vous à cliquer sur divers éléments de la page, et à changer les booléens, tous ensemble ou partiellement pour mieux comprendre le fonctionnement de ce paramètre. Mais encore une fois, il n’y a aucun problème à l’ignorer, ça fonctionne très bien par défaut, à tel point que ce paramètre est souvent passé sous silence dans les tutoriels qu’on peut trouver sur l’Internet.

10.6 Modifier le comportement des évènements

10.6.1 Arrêter la propagation

Nous avons vu que l’évènement se propage, par défaut de l’élément ciblé vers ses ancêtres. Oui mais que se passe-t-il si l’évènement est géré dans plusieurs gestionnaires d’évènement ? Ils sont forcément tous appelés, comme nous avons pu le voir dans l’exemple du chapitre précédent. Parfois ce n’est pas ce que nous voulons. Nous voudrions, par exemple, gérer le clic sur chaque élément indépendamment des autres. Il faudrait donc pouvoir interrompre la propagation de l’évènement. C’est possible avec la méthode `stopPropagation()` de l’objet `Event` dans une fonction de gestion de l’évènement (le callback, quoi !).

En reprenant l’exemple précédent, testez ce code :

```
document.addEventListener("click", function () {  
    console.log("document");  
});  
// Gestion du clic sur le paragraphe  
document.getElementById("para").addEventListener("click", function () {  
    console.log("paragraphe");  
});  
// Gestion du clic sur le bouton  
document.getElementById("btn").addEventListener("click", function (e) {  
    console.log("bouton");  
    e.stopPropagation();  
});
```

On peut constater dans la console que lorsqu’on clique sur le bouton, il n’y a plus propagation. En revanche, si on clique sur le paragraphe, on a bien propagation. Normal puisque la propagation n’est arrêtée qu’au niveau du bouton.

10.6.2 Supprimer le comportement par défaut

Certains évènements sont associés à un comportement par défaut. Typiquement, quand on clique sur un lien, le lien est suivi et le navigateur charge la page pointée par le lien. Ou bien le clic droit sur une page provoque l’affichage d’un menu contextuel. Il est possible d’annuler le comportement par défaut avec la méthode `preventDefault()` de l’objet `Event` :

```
<p>Vous voulez rire ? <a id="interdit"  
href="http://danstonchat.com/">Cliquez ici</a></p>
```

Avec le script :

```
document.getElementById("interdit").addEventListener("click", function (e)
{
    alert("Non non non. Continuez plutôt à lire le cours...");
    e.preventDefault(); // Annulation de la navigation vers le lien
});
```

Lors du clic sur le lien, un pop-up s'ouvre et la redirection n'a pas lieu.

10.7 Rappel sur le clonage

Rappelons que lors du clonage d'un nœud, les gestionnaires d'évènement ne sont pas clonés. Il faudra donc réassigner tous les écouteurs avec leur(s) callback(s) sur le clone.

11 La gestion des formulaires

11.1 Les propriétés des balises de formulaire

Au même titre que les autres éléments du DOM, ceux du formulaire sont accessibles et sélectionnables. Et de la même façon leurs attributs sont facilement lisibles ou modifiables. En revanche il y en a un certain nombre qui sont bien spécifiques aux formulaires. Il n'est donc pas inutile de les revoir.

Au niveau sémantique, rappelons qu'un formulaire contient un certain nombre de contrôles, qu'on appelle souvent des champs, chacun de ces contrôles pouvant encapsuler des options.

11.1.1 Accéder à la valeur d'un élément

La propriété `value` permet de définir une valeur pour différents éléments de formulaire, comme `input` ou `button`. Il suffit de lui assigner une chaîne de caractère pour qu'elle soit immédiatement répercutée à l'affichage.

Exemple : voici un champ textuel dont le contenu change selon qu'on lui donne le focus ou non. On utilise des écouteurs sur les événements `focus` et `blur` avec des fonctions anonymes comme `callback`.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title> Formulaire 01</title>
</head>
<body>
  <input id="text" type="text" size="60" value="Vous n'avez pas le focus !" />
  <script>
    const text = document.getElementById('text');
    text.addEventListener('focus', function(e) {
      e.target.value = "Vous avez le focus !";
    });
    text.addEventListener('blur', function(e) {
      e.target.value = "Vous n'avez plus le focus !";
    });
  </script>
</body>
</html>
```

11.1.2 Les booléens

En HTML, les attributs comme `disabled`, `checked` et `readonly` sont des booléens. Rappelons qu'en XML (et donc XHTML) ces propriétés doivent s'écrire sous la forme `<input checked="checked" />`. Cependant, en HTML, on peut utiliser la forme minimalisée `<input checked>`. En JavaScript, les propriétés correspondantes sont bien sûr des booléens également. Ainsi, pour désactiver un bouton, il suffit d'utiliser le code suivant :

```
const btn = document.getElementById('btn');
btn.disabled = true;
```

Il en sera de même pour toutes les propriétés représentant des attribut HTML booléens. Pour la propriété `checked`, on va utiliser une boucle pour parcourir tous les éléments associés.

Exemples :
Avec des boutons radio :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Formulaire 02</title>
  </head>
  <body>
    <label><input type="radio" name="check" value="1" /> Case
    n°1</label><br />
    <label><input type="radio" name="check" value="2" /> Case
    n°2</label><br />
    <label><input type="radio" name="check" value="3" /> Case
    n°3</label><br />
    <label><input type="radio" name="check" value="4" /> Case
    n°4</label><br />
    <input id="btn" type="button" value="Afficher la case cochée" />
    <script>
      const btn = document.getElementById('btn');
      btn.addEventListener('click', function() {
        const inputs = document.getElementsByTagName('input');
        for (let input of inputs) {
          if (input.type === 'radio' && input.checked) {
            alert('La case cochée est la n°' + input.value);
          }
        }
      });
    </script>
  </body>
</html>
```

On aurait pu simplifier ce code avec les sélecteurs CSS3. En effet, la vérification de type et le fait qu'une case soit cochée peuvent se faire directement dans le sélecteur. Comme c'est un ensemble de boutons radio, un seul peut être coché, on n'a donc plus besoin de passer par une boucle :

```
const btn = document.getElementById('btn');
btn.addEventListener('click', function() {
  const input = document.querySelector('input[type=radio]:checked');
  alert('La case cochée est la n°' + input.value);
});
```

Avec des cases à cocher :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document sans titre</title>
  </head>
  <body>
    <label><input type="checkbox" name="vehicule" value="Vélo">Vélo</label>
    <br />
    <label><input type="checkbox" name="vehicule" value="Moto">Moto</label>
```

```

    <br />
    <label><input type="checkbox" name="vehicule" value="Voiture">
Voiture</label>
    <br />
    <label><input type="checkbox" name="vehicule"
value="Camion">Camion</label>
    <br />
    <label><input type="checkbox" name="vehicule" value="Patins">Patins à
roulettes</label>
    <br /><br />
    <input id="btnVehicule" type="button" value="Afficher" />
    <script>
        const btnVehicule = document.getElementById('btnVehicule');
        btnVehicule.addEventListener('click', function() {
            let inputs =
document.querySelectorAll('input[type=checkbox]:checked');
            for (input of inputs) console.log(input.value);
        });
    </script>
</body>
</html>

```

11.1.3 Les listes déroulantes

Les listes déroulantes possèdent entr'autres les propriétés utiles `selectedIndex`, `selectedOptions` et `options`. La première nous donne l'identifiant de la valeur sélectionnée, la seconde retourne une collection des éléments sélectionnés dans le cas d'une liste à choix multiple, et la troisième retourne une collection des éléments `option` de la liste déroulante :

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document sans titre</title>
</head>
<body>
    <select id="list">
        <option>Sélectionnez votre pays</option>
        <option value="fr">France</option>
        <option value="us">USA</option>
        <option value="ja">Japon</option>
        <option value="cn">Chine</option>
    </select>
    <script>
        const list = document.getElementById('list');
        list.multiple = true; // La liste devient à sélection multiple
        list.addEventListener('change', function() {
            console.log(`Valeur sélectionnée :
${list.options[list.selectedIndex].value}`);
        });
        list.addEventListener('change', function() {
            for (option of list.selectedOptions) {
                console.log(option.value);
            }
        });
    </script>

```

```
</body>
</html>
```

Notez que `selectedIndex` est une propriété avec une valeur numérique. Dans le cas d'une liste à sélections multiples, elle ne contient que l'index de la première option sélectionnée. Pour obtenir l'ensemble des options sélectionnées, il faut utiliser la propriété `selectedOptions` et boucler dessus.

11.2 Les méthodes spécifiques aux formulaires

Un formulaire, au niveau du DOM, est un élément comme un autre. Il possède la propriété `elements` qui est une collection des contrôles appartenant à ce formulaire. Cette collection nous permet d'accéder à un contrôle par son nom (attribut `name`) ou par son indice (par ordre d'apparition dans le formulaire).

Exemple :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document sans titre</title>
</head>
<body>
  <form>
    <label><input type="checkbox" name="vehicule"
value="Vélo">Vélo</label>
    <br />
    <label><input type="checkbox" name="vehicule"
value="Moto">Moto</label>
    <br />
    <label><input type="checkbox" name="vehicule" value="Voiture">
Voiture</label>
    <br />
    <label><input type="checkbox" name="vehicule"
value="Camion">Camion</label>
    <br />
    <label><input type="checkbox" name="vehicule" value="Patins">Patins
à roulettes</label>
    <br /><br />
    <input id="btnVehicule" type="button" value="Afficher" />
  </form>
  <script>
    const form = document.querySelector('form');
    console.log(form.elements[0].name); // affiche 'vehicule'
    console.log(form.elements.vehicule.length); // affiche 5
    for (elt of form.elements) console.log(elt.value); // Affiche les 6
éléments du formulaire
  </script>
</body>
</html>
```

Il y a également deux méthodes importantes associées à l'objet formulaire, que nous allons détailler.

11.2.1 Méthode submit()

Cette méthode est utilisée pour soumettre le formulaire. C'est elle qui est appelée lorsque l'utilisateur clique sur le bouton d'envoi du formulaire, qui est le contrôle `input` de type `submit`.

En général, la soumission du formulaire se traduit par l'envoi de ses données à la ressource donnée dans l'attribut `action`, généralement un serveur qui traitera les données envoyées. Mais avant ce traitement, un évènement `submit` est déclenché. En mettant un écouteur sur le formulaire, on peut donc récupérer les données du formulaire avant leur envoi vers le serveur. Il est donc possible de vérifier les valeurs, voire de refuser l'envoi si on interdit le comportement par défaut de l'évènement avec `preventDefault()` !

Valider les données fournies par l'utilisateur avant de les transmettre au serveur est fondamental. Cependant, ne vous y trompez pas : c'est grandement insuffisant ! Si vous gérez la partie serveur, il est essentiel que le serveur fasse lui-même ses vérifications. Pourquoi ? Simplement parce que n'importe quel utilisateur peut par exemple désactiver JavaScript sur son navigateur, donc vos vérifications n'auront jamais lieu... À quoi ça sert alors ? À gagner du temps ! En effet, si c'est le serveur qui vérifie les données, alors il y a eu requête vers le serveur, calcul et vérifications par le serveur, puis retour du formulaire refusé. Beaucoup de temps et de bande passante utilisés pour rien, alors qu'on aurait pu faire la vérification en amont et ne soumettre le formulaire que lorsqu'il est conforme.

La validation du formulaire peut se faire au fur et à mesure de son remplissage comme à la soumission. C'est une simple question de choix. On peut même, et c'est recommandé, concilier les deux pour une plus grande interactivité.

Rappelez-vous également que pas mal de vérifications sont devenues inutiles avec HTML5 qui propose déjà une vérification à la source en fonction du type de contrôle utilisé. Mais rappelez-vous aussi que tout le monde n'a pas un navigateur compatible HTML5...

Sachez également qu'il est fréquent d'utiliser les *expressions régulières* -communément appelées *regex* ou *regexp*- pour faire des vérifications de formulaire. Je vous engage vivement à vous intéresser à ces expressions. Nous n'aborderons pas les regex tant leur compréhension peut remplir un livre complet...

11.2.2 Méthode reset()

Cette méthode est utilisée pour réinitialiser le formulaire. C'est elle qui est appelée lorsque l'utilisateur clique sur le bouton de réinitialisation, qui est le contrôle `input` de type `reset`.

Le clic sur ce bouton provoque un réaffichage du formulaire comme si vous veniez de charger la page, donc un formulaire vierge. Il génère aussi un évènement de type `reset` qu'on peut donc écouter et intercepter, voire empêcher de s'exécuter. Par exemple si on veut demander une confirmation avant d'effacer un formulaire de 200 champs...

```
form.addEventListener('reset', function (e) {  
    alert('Hors de question !');  
    e.preventDefault();  
});
```

11.3 Validation d'un formulaire

La validation des formulaires se fait automatiquement avec l'*API Validation* du HTML 5. Ainsi, le type des contrôles ainsi que certains attributs viennent imposer des contraintes sur ces mêmes contrôles. De ce fait, il devient quasiment inutile de faire une validation des formulaires en JS.

Il reste cependant possible de la faire, et on peut toujours s'appuyer sur la même API Validation.

Chaque contrôle, une fois récupéré, peut être interrogé sur son état (valide ou pas) grâce à deux fonctions qui se comportent presque de la même façon. Supposons que la variable input contienne un des contrôle du formulaire, alors on peut écrire :

```
let isValid = input.checkValidity();
```

Ou bien encore :

```
let isValid = input.reportValidity();
```

La différence entre `checkValidity()` et `reportValidity()` est que la seconde permet d'utiliser des messages personnalisés, que l'on peut fixer avec la méthode `setCustomValidity("message")`.

Voici un exemple simple de validation avec messages personnalisé :

```
<body>
  <form action="#" method="get">
    <div>
      <label for="name">Nom <span class="required">*</span></label>
      <input type="text" name="name" id="name" placeholder="Votre nom"
size="30" required minlength="2" maxlength="50">
    </div>
    <div>
      <label for="email">Courriel <span class="required">*</span></label>
      <input type="email" name="email" id="email" required>
    </div>
    <div>
      <label for="adh">N° adhérent <span class="required">*</span></label>
      <input type="text" name="adh" id="adh" required pattern="[0-9]{5}">
    </div>
    <div>
      <button type="submit">Envoyer</button>
    </div>
  </form>
  <script>
    // Gestion de la validité du formulaire avec l'API Validation
    document.querySelector("button").addEventListener("click", (evt) => {
      let isValid = true; // Un drapeau
      for (let input of document.querySelectorAll("input")) {
        if (input.name === "adh") { // Cas particulier du champ n° adhérent
          avec pattern
            // IMPORTANT ! Si un message d'erreur personnalisé est présent,
            // le contrôle est considéré comme invalide dans tous les cas !
            // Donc on remet ce message à vide.
            input.setCustomValidity("");
          if (input.value === "") { // champ vide => message d'erreur
            input.setCustomValidity("Veuillez remplir ce champ.");
          } else if (!input.validity.valid) { // champ a autre erreur =>
            autre message
              input.setCustomValidity("Ce champ ne doit contenir que 5
chiffres.");
            }
          }
        }
      }
      isValid = isValid && input.reportValidity();
    });
  </script>
```

```
    }  
    if (isValid) {  
        alert("Votre formulaire a bien été envoyé");  
    } else {  
        evt.preventDefault();  
    }  
    });  
</script>  
</body>
```

12 Animer les pages web

12.1 Gestion du temps

12.1.1 Action répétée

Supposons que je veuille un compteur de temps dans ma page. Par exemple pour afficher une horloge. Il faut bien que je puisse, à chaque seconde, faire avancer la trotteuse de cette horloge.

JavaScript met à notre disposition la fonction `setInterval(function, milliseconds)` qui prend deux paramètres : le premier est la fonction qui sera appelée, le second l'intervalle de temps entre chaque appel en millisecondes. Cette fonction retourne un identifiant d'action. Ainsi, pour mon horloge, j'aurai :

```
function addSecond() { // ajout d'une seconde à l'horloge}
const secondId = setInterval(addSecond, 1000);
```

Cette fonction permet donc de répéter une action à intervalles réguliers. Si on veut arrêter cette répétition, il suffit d'appeler la fonction `clearInterval(id)`. Cette fonction prend en paramètre l'identifiant d'action retourné par la fonction `setInterval()`.

12.1.2 Action retardée

De la même façon, on peut imaginer avoir besoin de lancer une action une seule fois au bout d'un temps donné. Par exemple, dans un jeu de questions, vous avez dix secondes avant de répondre. Au bout de ce temps, il devient impossible de répondre. En gros il nous faut un compte à rebours. JavaScript nous en propose un avec la fonction `setTimeout(function, milliseconds)`. Cette fonction utilise les mêmes paramètres que `setInterval()` et retourne le même type de valeur. Pour annuler le compte à rebours, il faut utiliser la fonction `clearTimeout(id)`.

12.1.3 Ajouter des paramètres

Parfois on aimerait bien pouvoir passer des paramètres à nos fonctions récurrentes ou retardées. C'est possible en les ajoutant simplement aux deux paramètres initiaux des fonction `setInterval()` et `setTimeout()`. Tous les paramètres supplémentaires seront passés à la fonction appelée. Ainsi :

```
setTimeout(maFonction, 2000, param1, param2);
```

Est équivalent à :

```
setTimeout(function() {
    maFonction(param1, param2);
}, 2000);
```

Cette dernière façon de faire est d'ailleurs la seule qui fonctionne sur Internet Explorer avant sa version 10...

12.2 Animer des éléments

Une animation n'est jamais qu'une succession d'images qui varient peu. Il s'agit donc bien de faire varier une propriété petit à petit d'un état initial à un état final et ce de façon régulière. On peut par exemple facilement faire un effet de fade in ou de fade out en jouant sur l'opacité d'une image.

Exemple : supposons qu'on a une image dans notre page web. Alors le code suivant la fait doucement presque disparaître :

```
<!DOCTYPE html>
```

```

<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document sans titre</title>
</head>
<body>
  
  <script>
    // On initialise le fading dès que la fenêtre a fini de se charger.
    window.addEventListener("load", fadeout);

    function fadeout() {
      const img = document.querySelector('img');
      const style = getComputedStyle(img);
      const opacity = parseFloat(style.opacity) - 0.01;
      img.style.opacity = opacity;
      if (opacity > 0.2) { // tant que l'opacité est supérieure à 0.2
        // On appelle la fonction qui fait diminuer l'opacité, donc
        elle-même
        setTimeout(fadeout, 50);
      }
    }
  </script>
</body>
</html>

```

On peut imaginer de la même façon déplacer un élément en changeant ses attributs `left` et `top`. Bref, on sait maintenant animer une page web.

NB : CSS3 a introduit toute une gamme de fonctionnalités simplifiant grandement les animations. Il n'y a donc plus particulièrement lieu d'utiliser JS pour ça, même si ça reste possible.

13 AJAX

13.1 Qu'est-ce que c'est ?

13.1.1 Concept

AJAX est l'acronyme de Asynchronous JavaScript And XML. C'est un ensemble de technologies destinées à faire des mises à jour rapides d'une partie du contenu d'une page HTML. AJAX est un sujet excessivement vaste, nous ne pourrions ici que l'effleurer.

AJAX fonctionne sur le principe client-serveur. Autrement dit, AJAX va initier une connexion au serveur pour lui réclamer des données, celui-ci va les lui fournir (normalement, s'il est gentil...), et AJAX va les utiliser pour les introduire dans la page web en modifiant le DOM.

L'avantage est que tout se passe de façon cachée, sans que l'utilisateur ne perçoive un quelconque changement de page par exemple.

Les utilisations sont multiples, par exemple l'auto-complétion d'un champ de recherche à la Google. Supposons que vous ayez un champ de recherche du nom d'un collègue. Dès que vous commencez à taper des caractères, AJAX va en « sous-marin » aller faire des requêtes auprès du serveur pour qu'il lui renvoie la liste des noms qui correspondent à ces premières lettres. Cette liste, une fois récupérée, peut-être affichée en propositions sélectionnables par l'utilisateur.

Ou bien on peut imaginer une page de blog dont une partie affiche des news qui se rafraîchissent toutes les 5 secondes : toutes les 5 secondes une requête AJAX est envoyée pour collecter les dernières news et les affiche dans la partie consacrée sur la page du blog.

13.1.2 Formats de données

Lors des échanges entre le client et le serveur, les données sont envoyées et reçues des deux côtés hypothétiquement. Il est donc nécessaire que les deux parlent le même langage. Les échanges se font la plupart du temps sous trois formats : texte, XML et JSON.

13.1.2.1 Format texte

Il n'y a pas grand-chose à en dire, c'est une succession de caractères qu'on peut directement utiliser. Si ce texte est en fait du HTML, on peut même le mettre directement dans notre DOM.

13.1.2.2 Format XML

XML signifie eXtended Markup Language. Il permet de structurer l'information grâce à des balises, comme le HTML, mais on peut utiliser les balises qu'on veut. XHTML est d'ailleurs une tentative d'implémentation de XML, abandonnée aujourd'hui.

Si on utilise la requête AJAX appropriée, les données une fois reçues seront automatiquement parcourues (grâce à un parseur), analysées et transformées en un DOM que l'on pourra parcourir de la même façon qu'on parcourt le DOM de notre page web.

Le gros avantage est donc la génération d'un DOM qu'on saura utiliser au mieux. L'inconvénient est que le XML est un langage très verbeux, qui a tendance à alourdir le transfert par un poids excessif.

13.1.3 Format JSON

JSON signifie JavaScript Object Notation. Il s'agit d'une représentation des données sous forme d'un objet littéral JavaScript (ce n'est pas tout à fait vrai, il y a de menues différences). De la même façon que pour le format XML, on peut utiliser un parseur pour transformer les données reçues directement en un objet JavaScript. La différence est que l'appel à ce parseur doit être fait par nous, il n'est pas automatique. Il faut pour cela utiliser la classe `JSON` qui met à notre disposition deux méthodes

fondamentales : `parse()` et `stringify()`. La première transforme donc les données reçues en objet JavaScript, la deuxième fait le contraire.

JSON est un format très pratique et condensé, reconnu par de très nombreux langages, en particulier PHP depuis sa version 5.2 qui met nativement à disposition des fonctions de gestion de ce format.

13.2 XHR

13.2.1 Quid ?

XHR est l'acronyme de XMLHttpRequest. C'est la façon la plus traditionnelle et la plus courante qui soit pour faire de l'AJAX.

À partir de maintenant, il va être nécessaire d'utiliser un serveur local. En effet, AJAX permet de faire des échanges entre un client (votre navigateur) et un serveur. Pas de panique, vous avez déjà tout ce qu'il faut. Toute cette partie sera donc occultée, et le code PHP vous sera fourni avec de brèves explications.

Il y a cependant un peu de configuration à faire. Éditez donc votre serveur Apache et vérifiez que la ligne suivante existe et n'est pas précédée d'un `#` :

```
LoadModule headers_module modules/mod_headers.so
```

Puis ajoutez à la fin les lignes suivantes :

```
# Accept cross-domain request
<IfModule mod_headers.c>
    Header always set Access-Control-Allow-Origin "*"
</IfModule>
```

Ceci afin de ne pas se heurter au problème de changement de serveur entre VSC et Apache.

Une autre alternative est de mettre au début de chaque script PHP la ligne suivante :

```
header('Access-Control-Allow-Origin: *');
```

13.2.2 L'objet XMLHttpRequest

Comme souvent, JavaScript met à notre disposition un objet pour gérer les requêtes XHR. XHR existe en deux versions, la seconde apparue au temps d'Internet Explorer 10 donc largement supportée à l'heure actuelle par tous les navigateurs. Nous utiliserons donc cette dernière.

L'utilisation de l'objet XHR se fait en trois étapes, comme toutes les requêtes :

- Préparation de la requête.
- Envoi de la requête.
- Réception des données.

Nous allons détailler ces étapes tout de suite.

13.2.3 Préparation des requêtes XHR

La première chose à faire, c'est d'avoir un objet XHR ! Nous allons donc le créer :

```
const xhr = new XMLHttpRequest();
```

Ensuite il faut préparer la requête avec la méthode `open(method, url, async, login, password)` qui prend cinq paramètres :

- La méthode d'envoi des données. Ce sont souvent les mêmes que pour un formulaire (donc `GET` ou `POST`), parfois `HEAD` et très rarement autre chose.
- L'URL à laquelle sera envoyée la requête au serveur, donc la ressource contactée.
- Un booléen facultatif indiquant si la requête est asynchrone ou pas. Par défaut, c'est `true`. Nous allons voir par la suite la différence.
- Les deux derniers arguments ne sont utiles que si une identification est nécessaire pour accéder à la ressource du serveur. Ce sont les paramètres d'identification habituels.

Habituellement, on n'utilise guère que les deux premiers paramètres :

```
xhr.open('GET', 'http://ma_ressource_serveur.fr/page.php');
```

Explications : cette ligne indique que la requête doit contacter le serveur sur l'adresse `http://ma_ressource_serveur.fr/page.php` en utilisant la méthode `GET`.

Une fois ces indications mises en place, on peut envoyer la requête au serveur avec la méthode `send()`. Cette méthode ne prend qu'un paramètre, qui sera `null` si on utilise la méthode `GET` ou `HEAD`, et qui contiendra les données envoyées si on utilise la méthode `POST` :

```
xhr.send(null);
```

Une telle requête asynchrone peut toujours être annulée en utilisant la méthode `abort()`, qui va instantanément arrêter toute activité de la requête. C'est utile si, en reprenant l'exemple de l'auto-complétion du champ de recherche d'un nom, l'utilisateur tape des lettres plus vite que le serveur ne répond à la requête : il est inutile d'attendre le retour du serveur puisqu'une autre requête va être envoyée dès qu'une nouvelle lettre est tapée.

13.2.3.1 Synchrone ou pas ?

Une requête asynchrone, une fois envoyée, laisse le script continuer de s'exécuter jusqu'à l'obtention complète de la réponse, moment auquel un événement est envoyé.

Une requête synchrone va bloquer le script jusqu'à obtention de la réponse. C'est bien sûr la plupart du temps inacceptable, l'utilisateur ne va pas attendre bêtement devant son ordinateur parce que vous avez fait une requête en sous-marin. D'autre part, l'utilisation de requêtes synchrone a été dépréciée (justement à cause de son impact négatif sur la navigation de l'utilisateur).

L'utilisation de requêtes asynchrones est donc grandement préférée dans la plupart des cas. Et c'est pourquoi ce paramètre n'est quasiment jamais utilisé.

13.2.3.2 HEAD, GET ou POST ?

`HEAD` n'est pas utilisé pour envoyer des données, mais uniquement pour en recevoir. Plus précisément, vous recevrez seulement les entêtes de la ressource contactée (son *header*, d'où le terme `HEAD`). Ce n'est réellement utile que pour savoir si la ressource existe bien sur le serveur.

`GET` et `POST` permettent toutes deux d'envoyer des données. Cependant, la règle est que `GET` ne doit être utilisé que pour recevoir des données depuis le serveur. `POST` sera utilisé pour envoyer des données destinées à modifier des données sur le serveur.

En gros, si vous ne voulez que charger un fichier, vous utiliserez `GET`, passant en paramètre le nom du fichier par exemple. En revanche, si vous avez rempli un formulaire de mise à jour de vos données personnelles sur le serveur, vous utiliserez `POST`. Vous pouvez voir toutes les méthodes utilisables dans une requête HTTP et leurs différences sur ce lien : https://www.w3schools.com/tags/ref_httpmethods.asp

Sachez également que `POST` provoque un comportement particulier des navigateurs (du moins dans la théorie) : si vous rechargez la page (touche `F5` par exemple), la requête est de nouveau soumise. C'est automatique pour `GET`, mais provoquera pour `POST` l'apparition d'une fenêtre de confirmation. Et c'est bien logique, puisque `POST` est censé être utilisé pour modifier des données sur le serveur. Imaginez que vous ayez donné l'ordre à votre banque de virer 1000€ sur le compte d'un ami, en rechargeant la page vous provoquez un deuxième virement ! C'est votre ami qui sera content ! Le banquier, moins...

13.2.3.3 Envoyer des paramètres

Les paramètres sont toujours envoyés sous la même forme. C'est une chaîne de caractères contenant des paires "clé=valeur" séparées par desesperluettes `&`. Que nous utilisions `GET` ou `POST` ne change rien à cette chaîne. Ce qui va changer, c'est où la mettre. Cette chaîne s'appelle une *query string* en anglais.

Avec `GET`, vous allez les mettre directement derrière l'URL après avoir ajouté un point d'interrogation dans la méthode `open()` :

```
xhr.open('GET',  
'http://ma_ressource_serveur.fr/page.php?key1=value1&key2=value2');  
xhr.send(null);
```

Dans cet exemple, je passe deux paramètres, le premier s'appelant `key1` et ayant la valeur `value1`, le second `key2` ayant la valeur `value2`.

Avec `POST` vous allez les mettre dans la méthode `send()`, non sans avoir au préalable indiqué que vous envoyez des données de formulaire (même si ce n'est pas le cas !) avec la méthode `setRequestHeader(header, value)` qui va modifier les headers de votre page :

```
xhr.open('POST', 'http://ma_ressource_serveur.fr/page.php');  
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");  
xhr.send('key1=value1&key2=value2');
```

Cette façon de faire n'est cependant pas optimale, car elle nécessite de se souvenir de cette phase de changement des headers, ce qui est complexe.

JavaScript propose une autre alternative bien plus agréable avec l'objet `FormData`. Cet objet, une fois créé, permet d'ajouter automatiquement des couples clé-valeur via son unique méthode `append(key, value)`. Ceci fait, il suffit de passer l'objet en paramètre de la méthode `send()` sans plus se préoccuper des headers.

Notre code s'écrit donc :

```
xhr.open(POST, 'http://ma_ressource_serveur.fr/page.php');  
let formData = new FormData();  
formData.append('key1', 'value1');  
formData.append('key2', 'value2');  
xhr.send(formData);
```

Encore plus fort, l'objet `FormData` peut être instancié en lui passant directement un élément `form` du DOM, ce qui évite les fastidieux ajouts via `append()`.

13.2.3.4 Limiter le temps imparti

Parfois on peut souhaiter limiter le temps maximal que peut prendre une requête. Une première solution est d'utiliser la méthode `abort()` dans un `setTimeout()`. Mais plus simplement depuis XHR

version 2 il existe la propriété `timeout` qui prend pour valeur le nombre de millisecondes maximum que vous donnez à la requête pour aboutir. Une fois ce temps écoulé, la requête se terminera.

13.2.4 Réception des données XHR

Après réception du résultat de la requête, les données renvoyées par le serveur sont stockées dans la propriété `responseText` de l'objet XHR sous forme textuelle. En cas de requête asynchrone (la majorité du temps donc), on peut suivre la vie de la requête grâce à des événements. Historiquement, le seul événement était `readystatechange`. Mais cet événement se déclenche à chaque changement d'état de la requête, il faut donc connaître ces différents états, qui sont représentés par des constantes numériques de la classe `XMLHttpRequest`, mais surtout le callback est une fonction complexe qui comporte plusieurs possibilités. Je vous donne quand même les états possibles :

Constante	Valeur	Description
UNSENT	0	L'objet XHR a été créé, mais pas initialisé (la méthode <code>open()</code> n'a pas encore été appelée).
OPENED	1	La méthode <code>open()</code> a été appelée, mais la requête n'a pas encore été envoyée par la méthode <code>send()</code> .
HEADERS_RECEIVED	2	La méthode <code>send()</code> a été appelée et toutes les informations ont été envoyées au serveur.
LOADING	3	Le serveur traite les informations et a commencé à renvoyer les données. Tous les en-têtes des fichiers ont été reçus.
DONE	4	Toutes les données ont été réceptionnées.

Depuis la version 2 de XHR, il existe pas moins de 8 événements, si on compte `readystatechange`. Pourquoi autant ? Parce que XHR1 ne permettait pas un suivi suffisamment correct de la requête.

Voici la liste de ces événements :

Évènement	Déclencheur
<code>loadstart</code>	La collecte des données a commencé (lors de l'appel à <code>send()</code>).
<code>progress</code>	Transmission des données.
<code>abort</code>	La collecte des données est arrêtée (par exemple avec <code>abort()</code>)
<code>error</code>	La collecte des données a échoué.
<code>load</code>	La collecte des données a réussi.
<code>timeout</code>	Dépassement du temps imparti à la collecte.
<code>loadend</code>	La collecte des données est finie (réussite ou échec).

13.2.5 Gestion des erreurs

Un échange de données avec un serveur peut échouer pour plusieurs raisons : le serveur n'est pas joignable, ou bien il l'est mais la ressource demandée n'existe pas ou n'est pas accessible, il y a une panne de réseau en plein transfert... L'évènement `error` permet déjà de savoir si notre requête a donné un résultat ou pas, c'est un bon point de départ. Si cet événement apparaît, c'est que nous n'avons pas contacté le serveur. Ensuite, le serveur renvoie un statut sous forme de valeur numérique.

Si ce statut est une valeur comprise entre 200 (inclus) et 400 (exclus), alors notre requête a abouti. Sinon, c'est qu'il y a eu une erreur au niveau du serveur. Par exemple, la page demandée n'existe pas (la fameuse erreur 404 que vous avez tous rencontrée). Le statut est stocké dans la propriété `status` de la requête (l'objet XHR pour être plus précis), et son explication dans la propriété `statusText`, on peut donc penser à un code de ce genre pour une gestion succincte des erreurs :

```
const xhr = new XMLHttpRequest();
xhr.open('GET', myURL);
xhr.addEventListener('load', function () {
    if ((xhr.status >= 200) && (xhr.status < 400)) { // Le serveur a réussi
à traiter la requête
        console.log(xhr.responseText);
    } else {
        // Affichage des informations sur l'échec du traitement de la
requête
        console.error(xhr.status, xhr.statusText);
    }
});
xhr.addEventListener('error', function () {
    // La requête n'a pas réussi à atteindre le serveur
    console.error("Erreur réseau");
});
xhr.send(null); // Envoi de la requête
```

13.2.6 Exemples

Deux exemples pour poser les choses.

13.2.6.1 Exemple 1 : le chaton

Nous allons afficher une image dans notre page, image récupérée par AJAX. Ici, l'image se trouve dans le même répertoire que le script (il faut donc bien que l'image existe !).

13.2.6.1.1 Partie HTML

Le code est on ne peut plus simple : il n'y a qu'une `div` vide attendant qu'on y mette l'image !

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ajax natif</title>
</head>
<body>
    <div class="resultat"></div>
    <script> </script>
</body>
</html>
```

13.2.6.1.2 Partie JS

Le script (soit à sourcer dans la balise `script`, soit à mettre directement dedans) :

```
const xhr = new XMLHttpRequest();
const url = 'chaton.jpg';
xhr.open('GET', url);
xhr.responseType = 'blob'; // On reçoit du binaire
// Gestion événementielle de la requête
xhr.addEventListener('load', () => { // On a reçu une réponse du serveur,
mais laquelle ?
```

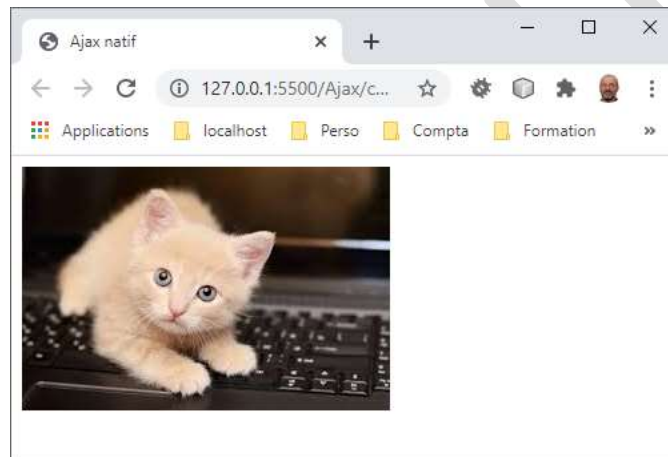
```

    if (xhr.status >= 200 && xhr.status < 400) { // On a nos données
        const img = document.createElement('img');
        img.src = URL.createObjectURL(xhr.response);
        document.querySelector(".resultat").appendChild(img);
    } else { // On a quelque chose mais pas les données
        document.querySelector(".resultat").textContent = `Erreur
${xhr.status} : ${xhr.statusText}`;
    }
});
xhr.addEventListener('error', (e) => { // Le serveur n'a pas répondu
    document.querySelector(".resultat").textContent = 'Erreur : un problème
réseau est survenu.';
});
xhr.send(null);

```

13.2.6.1.3 Résultat

Vous devriez avoir l'image qui apparaît dans votre page (instantanément, vu que le serveur devrait répondre TRÈS vite).



13.2.6.2 Exemple 2 : mon API

Nous allons faire un petit code qui va soumettre un formulaire contenant un simple champ texte à notre serveur, qui répondra en retournant simplement le texte qu'on lui a soumis. Ce texte sera ensuite affiché dans la console de l'interpréteur JavaScript. Pour simplifier on ne gèrera pas les erreurs.

L'URL contactée est donc une URL locale. Si vous n'avez pas de serveur, ou ne savez pas le configurer, vous pouvez utiliser cette URL qui fonctionnera de la même façon : <http://cam.ldnr.fr/~herbertcaffarel/js/test.php>.

13.2.6.2.1 Côté serveur

Nous allons écrire une page PHP qui va simplement renvoyer sous forme de texte les données reçues depuis notre client en POST avec la clé `myText`. Cette page PHP sera positionnée dans la racine de votre serveur Apache, dans un sous-répertoire nommé `coursJS`. Cette page se nommera `test.php` et contiendra :

```

<?php
header('Access-Control-Allow-Origin: *');
echo 'reçu : ', $_POST['myText'];
?>

```

Explication : la première et la dernière ligne indiquent qu'on écrit un script PHP. La seconde ligne autorise les requêtes cross-domain, et la troisième ligne écrit simplement le texte « Reçu : » suivi du

contenu du champ `myForm` reçu du formulaire. Ce contenu est donc renvoyé par le serveur et sera récupéré par notre XHR.

13.2.6.2.2 Partie HTML

Le code de notre page web, simple et sans fioriture :

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>My API</title>
</head>
<body>
  <form id="myForm">
    <label>Votre texte
      <input id="myText" name="myText" type="text" placeholder="Test
! Un, deux, un, deux !">
    </label>
    <input type="button" value="go" id="btn">
  </form>
  <script></script>
</body>
</html>
```

Un formulaire contenant un champ texte éditable et un bouton (qui n'est pas un bouton de soumission pour éviter de vider la console à chaque soumission) pour y accrocher un évènement.

13.2.6.2.3 Côté JavaScript

Voici le code à utiliser dans la balise `script` de notre page HTML :

```
// Gestion événementielle du clic sur le bouton => appel requête ajax
document.getElementById('btn')
  .addEventListener('click', sendAjaxRequest);
document.getElementById('myText')
  .addEventListener('keydown', (evt) => { // On surveille le clavier
    if (evt.keyCode == 13) {
      sendAjaxRequest(evt); // Si touche return en lance AJAX
      evt.preventDefault(); // Sinon formulaire envoyé par défaut !
    }
  });

// Fonction effectuant la requête ajax
function sendAjaxRequest() {
  const xhr = new XMLHttpRequest(), // L'objet encapsulant la requête
    url = 'http://cam.ldnr.fr/~herbertcaffarel/js/test.php'; // L'URL
  contactée
  // Création de la requête en POST
  xhr.open('POST', url);
  // Gestion événementielle du retour de la requête
  // => affichage du texte retour en console
  xhr.addEventListener('load', function () {
    console.log(xhr.responseText);
  });
  // Collecte des données du formulaire et passage des paramètres
  // dans la query string
  const form = new FormData(document.getElementById('myForm'));
  // Envoi de la requête ajax
```

```
xhr.send(form);  
}
```

Explications : tout se passe dans le gestionnaire d'évènement du clic sur le bouton ou lors de l'appui sur la touche *entrée* du clavier après saisie.

La première ligne fait :

- Sélection du bouton par son `id`.
- Ajout d'un écouteur sur ce contrôle, surveillant le clic.
- Association d'une fonction callback `sendAjaxRequest` au clic.

La seconde ligne fait :

- Sélection du champ par son `id`.
- Ajout d'un écouteur surveillant l'appui sur une touche
- Association d'un callback qui :
 - Vérifie si la touche appuyée est la touche *entrée*.
 - Si c'est le cas appelle la fonction `sendAjaxRequest` et empêche la l'évènement de faire son travail (à savoir : soumettre le formulaire !), sinon la page serait rechargée, ce qu'on ne veut pas puisqu'on perd la console.

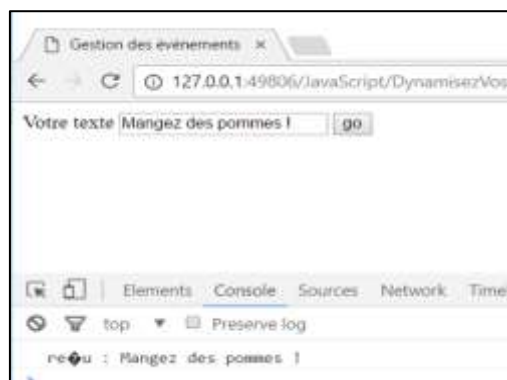
La suite est le contenu de la fonction `sendAjaxRequest`. On y trouve :

- Instanciation d'un objet XHR, et stockage dans une variable de l'URL de notre ressource serveur.
- Préparation de la requête qui sera envoyée en `POST` à notre URL.
- Ajout d'un écouteur sur la fin de récupération des données issues du serveur, dans lequel le callback est un simple affichage dans la console des données reçues. On ne gère pas les erreurs pour plus de lisibilité, mais nous avons vu comment les traiter juste avant.
- Instanciation d'un objet `FormData` avec le formulaire récupéré de notre page web pour le remplir automatiquement
- Envoi de la requête.

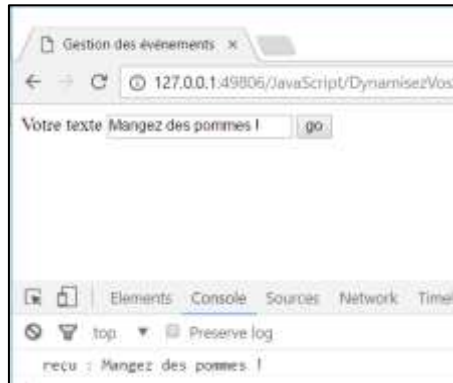
13.2.6.2.4 Résultat

Après avoir rempli le champ, en cliquant sur le bouton, la requête est construite et soumise au serveur, qui répond en retournant un simple texte. Dès que la réception est terminée (ce qui est immédiat dans ce cas, du moins si votre serveur est actif et votre script PHP au bon endroit), le callback affichant ce texte reçu est appelé.

Et on obtient :



Mais c'est quoi ce caractère ésotérique dans ma console ? Simplement un problème d'encodage des caractères. Il se trouve que j'ai écrit le script PHP dans un autre éditeur, et je suis tombé dans le piège classique : cet éditeur est configuré pour encoder les caractères en *iso-8859-1*. Or tout est configuré par défaut pour l'encodage *UTF-8*, que ce soit VSC ou le serveur Apache. Bien fait pour moi, et bonne leçon pour vous : pensez toujours à configurer tout votre environnement de développement en *UTF-8*. Il suffit de réenregistrer le fichier `test.php` en *UTF-8* pour que tout rentre dans l'ordre :



14 Des promesses, toujours des promesses

14.1 Les promesses en JavaScript

Nous avons vu qu'il était simple de passer un traitement à une fonction. Il suffit de fournir en paramètre... une fonction. On l'utilise avec les callback, que nous utilisons intensément dans la gestion événementielle ou encore dans les requêtes AJAX.

Mais que se passe-t-il si on a un empilement de callback ? Par exemple une partie du résultat de notre requête AJAX sert pour une autre requête AJAX, qui elle-même paramètre une autre requête... Dans ce cas, le code récupéré par notre première requête va être exploité dans le callback, qui va lui-même créer la seconde requête AJAX dont le résultat, exploité dans son callback, va créer une troisième requête AJAX... Notre code s'empile, et par le jeu de l'indentation, il se retrouve de plus en plus poussé vers la droite de l'écran, jusqu'à devenir invisible. Les anglais ont appelé cette forme la *pyramid of doom*, ou encore *callback hell*.

C'est bien sûr inacceptable. Une première possibilité pour linéariser est d'utiliser des callback nommés. Mais définir une fonction pour ne l'utiliser qu'une seule fois n'est pas des plus pertinents, même si c'est acceptable. Aussi ES6 propose une autre solution : les promesses (en anglais promise).

Le principe consiste simplement à avoir un objet qui gère seul les traitements asynchrones. Imaginez le principe actuel : lorsqu'un traitement se fait de façon asynchrone (autrement dit : on ne sait jamais quand !), on « prévoit » de lui faire faire quelque chose en réponse. En gros, on s'attend à ce que le traitement produise un résultat qu'on exploitera. Maintenant, imaginez que ce traitement nous fasse une promesse : « promis-craché-juré, quand j'ai un résultat je te préviens ». Et avec cette promesse, je peux « prévoir l'avenir ». Je vais donc me dire : « si le résultat correspond à ce que j'attends, je fais ceci, sinon je fais cela. Et au pire, s'il n'y a pas de résultat, je fais encore autre chose ».

C'est ce que proposent les promesses en JS. Une promesse est un objet de type `Promise` dont le constructeur prend en paramètre deux fonctions (couramment appelées `resolve` et `reject`) et dont le corps est appelé executor.

Lorsqu'on crée une promesse, le corps de la promesse est immédiatement exécuté, et à l'issue du traitement (qui rappelons-le est généralement asynchrone), la promesse nous informe que le traitement a bien eu lieu (on dit que la requête est résolue ou en anglais resolved), ou au contraire que quelque chose s'est mal passé et que la promesse est brisée ou rejetée (en anglais rejected).

La promesse étant un objet, elle propose des méthodes, dont deux en particulier nous intéressent : la méthode `then()` et la méthode `catch()`, qui retournent toutes deux une promesse.

La méthode `then()` permet de traiter la réponse issue de la promesse quand réponse il y a (promesse résolue), la méthode `catch()` permettant quant à elle de traiter une promesse rejetée. Notons qu'on peut parfaitement mettre en second paramètre de `then()` la fonction de traitement des erreurs. En gros, les codes suivants sont identiques :

```
promise.then(fonctionOK(resp), fonctionErreur(err));
promise.then(fonctionOK(resp))
    .catch(fonctionErreur(err));
```

En fait, pas tout à fait la même chose. En effet, dans le premier cas, si `fonctionOK()` lève une exception, elle NE sera PAS capturée, alors que dans le second cas, le `catch()` capturera toutes les exceptions, que ce soit une promesse rejetée ou une erreur levée par `fonctionOK()`. La seconde écriture est donc de loin préférable.

Voici donc à quoi ressemblerait notre chaton avec un traitement « façon promesse » :

```
let promise = new Promise(function (resolve, reject) {
  // une XHR pour télécharger une image locale
  const request = new XMLHttpRequest();
  request.open('GET', 'chaton.jpg'); // L'image doit exister...
  request.responseType = 'blob';
  // Quand la requête est complétée, on vérifie que tout s'est bien passé
  request.onload = function () {
    if (request.status >= 200 && request.status < 400) {
      // OK, tout va bien, la promesse est résolue.
      // On retourne donc vers le code appelant en indiquant une
      // promesse résolue et en passant en argument la réponse
      // obtenue
      resolve(request.response);
    } else {
      // il s'est passé quelque chose. On a reçu une réponse, mais
      // pas celle attendue. La promesse est donc rejetée, et une
      // erreur est remontée
      reject(new Error("L'image ne s'est pas chargée correctement.
Code d'erreur : " + request.statusText));
    }
  };
  request.onerror = function () {
    // Ici, on n'a même pas reçu de réponse. La promesse est donc
    // également rejetée, avec une erreur différente
    reject(new Error('Un problème réseau est survenu.'));
  };
  // Envoi de la requête
  request.send();
});
// La promesse est faite, je n'ai plus qu'à gérer les retours (résolue avec
then, rejetée avec catch)
promise
  .then(function (response) { // response est la valeur retournée par
    // resolve
    // on récupère les données récupérées par l'executor de la promesse
    // dans response
    let imageURL = window.URL.createObjectURL(response);
    // On crée un élément image qu'on insère dans le body
    let myImage = document.createElement("img");
    myImage.src = imageURL;
    document.body.appendChild(myImage);
  })
  .catch(function (err) { // err est la valeur retournée par reject
    // La promesse est rejetée, on récupère l'erreur et on l'affiche
    console.log(err);
  });
});
```

Si l'image locale `chaton.jpg` se trouve bien avec votre fichier html, l'image sera affichée dans la page html (traitement dans le `then()`), sinon la promesse sera rejetée et donc traitée par le `catch()` qui affichera une erreur dans la console :

```
Error: L'image ne s'est pas chargée correctement. Code d'erreur : Not Found
at XMLHttpRequest.request.onload (02-ajax_promise.html:30)
```


Le gros intérêt des promesses est qu'on peut les chaîner. En effet, `then()` et `catch()` retournent une promesse. On peut ainsi enchaîner les traitements asynchrones tout en gardant un code parfaitement lisible puisqu'il n'y a plus d'indentations multiples.

Notons que dans notre exemple précédent, les fonctions `then()` et `catch()` ne retournent rien, donc retournent `undefined`. C'est important si on souhaite chaîner d'autres traitements ! À ce sujet, je vous engage vivement à lire cette page : <https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>.

Voici un exemple d'école pour chaîner 2 traitements asynchrones, le second utilisant les données du premier.

Dans un premier temps on se dote d'une fonction `ajaxGet(url)` qui prend une url (sous forme de String) en paramètre et qui retourne une promesse. Ceci permet de simplifier l'utilisation des XHR en les encapsulant dans une petite API (d'ailleurs, vous pouvez vous amuser à faire la méthode `ajaxPost(url)` et à mettre ces deux fonctions dans un fichier `ajax.js`, vous l'utiliserez souvent ! Vous pouvez même faire un objet...) :

```
// Exécute un appel AJAX GET
// Prend en paramètres l'URL cible et retourne une promesse
function ajaxGet(url) {
    return new Promise(function(resolve, reject) {
        // Création d'une requête HTTP
        const xhr = new XMLHttpRequest();
        // Requête HTTP GET asynchrone car le 3ème paramètre est true
        xhr.open("GET", url, true);
        // Gestion de l'événement indiquant la fin de la requête
        xhr.addEventListener("load", function() {
            if (xhr.status >= 200 && xhr.status < 400) { // Le serveur a
réussi à traiter la requête
                // On retourne une promesse résolue
                resolve(xhr.responseText);
            } else {
                // échec du traitement de la requête => on retourne une
promesse rejetée
                reject(new Error(xhr.status + " " + xhr.statusText + " " +
url));
            }
        });
        xhr.addEventListener("error", function() {
            // La requête n'a pas réussi à atteindre le serveur => on
retourne une promesse rejetée
            reject(new Error("Erreur réseau avec l'URL " + url));
        });
        // Envoi de la requête
        xhr.send(null);
    });
}
```

Voici maintenant une utilisation cumulée de deux utilisations de `ajaxGet()`. En utilisant des callbacks, on aurait obligatoirement un second appel à `ajaxGet()` dans le callback du premier appel. En utilisant les promesses, on linéarise tout ça. L'astuce consiste à ce que le premier traitement (le premier `then()`) retourne lui-même une promesse, autrement dit... le résultat d'un appel à `ajaxGet()`.

Ici nous consommons une API qui est celle de *JC Decaux*, l'entreprise qui met des stations de vélos à louer dans toutes les grandes villes de France. Elle met à notre disposition un certain nombre d'informations sous forme JSON par simple interrogation d'une URL. Vous pouvez aller voir cette API sur ce lien : <https://developer.jcdecaux.com/#/opendata/vls?page=getstarted> (notez qu'il faut s'inscrire, c'est gratuit, et ça vous permettra en plus d'avoir une clé personnelle pour utiliser l'API). Notre but est d'afficher les informations de la première station de vélo de la ville de Créteil. Nous allons donc dans un premier temps récupérer la liste de toutes les stations, puis dans un second temps récupérer les informations spécifiques de la première station de la liste grâce à sa propriété `number`.

Oui, il serait plus simple de récupérer directement ces informations dès la première requête parce qu'on peut, mais pour l'exemple je fais bien 2 requêtes.

```
function showFirstStation() {
    // On consomme une API : récupération de toutes les stations de
    // location de vélo de Créteil. La méthode ajaxGet retourne donc
    // une promesse, résolue ou rejetée
    let promesseStations =
    ajaxGet("https://api.jcdecaux.com/vls/v1/stations?contract=Creteil&apiKey=e123ba7b0bb7819dd99efe71600461a132305da2");
    // Une fois la promesse résolue récupérée, on peut travailler sur le
    // résultat récupéré
    promesseStations
        .then(function(response) {
            // on récupère les données récupérées par la promesse
            let stations = JSON.parse(response);
            // Pour debug on affiche toutes les stations en console
            stations.forEach(station => console.log(station));
            // On récupère maintenant toutes les données de la première
            // station de la liste
            // toujours avec ajaxGet qui retourne une promesse
            let promessePremiereStation =
            ajaxGet("https://api.jcdecaux.com/vls/v1/stations/" + stations[0].number +
            "?contract=creteil&apiKey=e123ba7b0bb7819dd99efe71600461a132305da2");
            // Et on retourne cette promesse ! ce qui permet d'enchaîner
            // les then()/catch()
            return promessePremiereStation;
        })
        .then(function(resp) {
            // On récupère les données fournies par la promesse résolue
            let station = JSON.parse(resp);
            for (let info in station) {
                let div = document.createElement("div");
                div.innerHTML = info + ": " + station[info];
                document.body.appendChild(div);
            }
        })
        .catch(function(err) {
            // La promesse est rejetée (la première ou la seconde), on
            // récupère l'erreur et on l'affiche
            // On récupérerait même une erreur levée par l'une des deux
            // fonction then() !
            console.log(err);
        });
}
showFirstStation();
```

On voit qu'il est très simple d'enchaîner les `then()/catch()` moyennant de penser à retourner une promesse lors des utilisation de ces fonctions.

Remarquez aussi qu'il est tout à fait possible d'enchaîner des `then()` après des `catch()` si les `catch()` retournent des promesses. Bref, tout cela est fort logique. Il y a bien sûr énormément de choses à dire sur les promesses, mais vous avez maintenant un premier aperçu.

Pour aller plus loin, je vous convie à lire les documents suivants :

Bien sûr la documentation de MDN :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise

Une excellente présentation très complète des promesses : <http://javascript.info/async> (en anglais).

Pour aller encore plus loin et rentrer dans les détails qui tuent : <https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>.

14.2 L'API Fetch

En même temps que les promesses, ES6 a introduit une API destinée à remplacer AJAX, ou du moins à simplifier son utilisation en utilisant justement les promesses.

Cette API a pour centre la fonction `fetch(uri)`. Cette fonction retourne une promesse, qu'on peut donc aisément manipuler avec `then/catch`. Le contenu de la promesse résolue est un objet de type `Response`, lequel propose des méthodes de résolution du contenu de la réponse en texte, en fichier binaire (blob), en FormData (un dictionnaire associé à un formulaire) et d'autres. Comme toujours, allez voir la documentation.

Voici notre chat avec l'API Fetch :

```
const url = 'chaton.jpg';
fetch(url)
  .then(response => {
    if (response.ok) return response.blob();
    else throw new Error(`Erreur ${response.status} :
${response.statusText}`);
  })
  .then(blob => {
    const img = document.createElement('img');
    img.src = URL.createObjectURL(blob);
    document.querySelector(".resultat").appendChild(img);
  })
  .catch(e => document.querySelector(".resultat").textContent =
`Erreur : un problème réseau est survenu (${e}).`);
```

14.3 Les promesses avec ES7

ES7 a doté JS d'un sucre syntaxique permettant d'utiliser encore plus simplement les promesses. En effet, le problème majeur des promesses, c'est qu'elles ne permettent pas vraiment de sortir du principe du callback, à savoir l'appel asynchrone, avec une portion de code qui s'exécute... on ne sait pas quand. Alors qu'on a pris l'habitude de programmer séquentiellement, ce qui fait que la réflexion asynchrone perturbe notre beau modèle mental.

ES7 présente donc les instructions `async/await`. Il est possible de déclarer qu'une fonction fait de l'asynchrone en la préfixant par le seul mot-clé `async`. Dès lors il devient possible d'utiliser à l'intérieur

le mot-clé `await` à placer avant l'appel à une fonction retournant une promesse, et qui va littéralement mettre ce bout de code en pause jusqu'à ce que le résultat de la promesse arrive et mette à disposition le résultat de cette promesse (les données recueillies par la promesse résolue). Attention, j'ai bien dit : ce bout de code ! Le reste du programme se déroule normalement, mais ce bout de code se met en pause, et dès que la promesse arrive (résolue ou rejetée) le code reprend. Ce qui permet d'écrire notre code de façon linéaire et séquentielle comme on en a l'habitude. En cas de promesse rejetée, l'erreur est propagée. Ce qui signifie bien sûr qu'on retrouve le mécanisme habituel du `try/catch`, ce qui est beaucoup plus dans nos habitudes également.

J'insiste sur le fait que `await` ne peut être utilisé que dans une fonction `async`. On ne peut donc pas l'utiliser dans le code principal, au niveau duquel il devient nécessaire de traiter les promesses de façon plus traditionnelle.

Voici notre chat avec `async/await` :

```
async function getAndShowCat(url) {
  try {
    const response = await fetch(url);
    if (!response.ok) throw new Error(`Erreur ${response.status} :
    ${response.statusText}`);
    const blob = await response.blob();
    const img = document.createElement('img');
    img.src = URL.createObjectURL(blob);
    document.querySelector(".resultat").appendChild(img);
  } catch (e) {
    document.querySelector(".resultat").textContent = `Erreur : un
    problème réseau est survenu (${e}).`;
  };
};
getAndShowCat('chaton.jpg');
```

Voici ce que donnerait le code d'affichage d'une station de vélo avec l'utilisation de `async/await` :

```
async function showFirstStation() {
  // On consomme une API : récupération de toutes les stations de
  // location de vélo de Créteil. La méthode fetch retourne donc
  // une promesse résolue sous forme d'objet de type Response ou
  // une erreur (comme exception en Java)
  try {
    // await attend le retour de cette promesse et fournit
    // le résultat (données)
    // En cas d'erreur, il propage l'erreur dans le try qui se branche
    // donc sur le catch
    let resultPromesseStations = await fetch(
    "https://api.jcdecaux.com/vls/v1/stations?contract=Creteil&apiKey=e123ba7b0
    bb7819dd99efe71600461a132305da2");
    // on dé-JSONifie les données récupérées par la promesse
    let stations = await resultPromesseStations.json(); // retourne une
    promesse !
    // Pour debug on affiche toutes les stations en console
    stations.forEach(station => console.log(station));
    // On récupère maintenant toutes les données de la première station
    de la liste
    // toujours avec fetch qui retourne une promesse résolue
    // et await qui fournit les données (ou propage une erreur)
```

```

    let resultPromessePremiereStation = await
    fetch("https://api.jcdecaux.com/vls/vl/stations/" +
        stations[0].number +
        "?contract=creteil&apiKey=e123ba7b0bb7819dd99efe71600461a132305da2");
    // On dé-JSONifie les données fournies par la promesse résolue
    let station = await resultPromessePremiereStation.json(); //
    retourne une promesse !
    // On parcourt et affiche les infos de la station dans la page
    for (let info in station) {
        let div = document.createElement("div");
        div.innerHTML = info + ": " + station[info];
        document.body.appendChild(div);
    }
} catch (err) {
    console.log(err);
}
}

```

On voit que la fonction est plus courte (une dizaine de lignes gagnées), mais surtout beaucoup plus lisible : notre code s'enchaîne comme on en a l'habitude, comme si tout se passait de façon synchrone, sauf que `await` met le travail en pause directement dans le code. Pour nous, c'est transparent et bien plus accessible. En plus, on retrouve notre mécanisme habituel de gestion des erreurs avec le couple `try/catch`.